IT UNIVERSITY OF COPENHAGEN

DOCTORAL THESIS

Delilah: Efficient eBPF Offload for Integrated Data Pipelines on Computational Storage

Author:
Niclas HEDAM

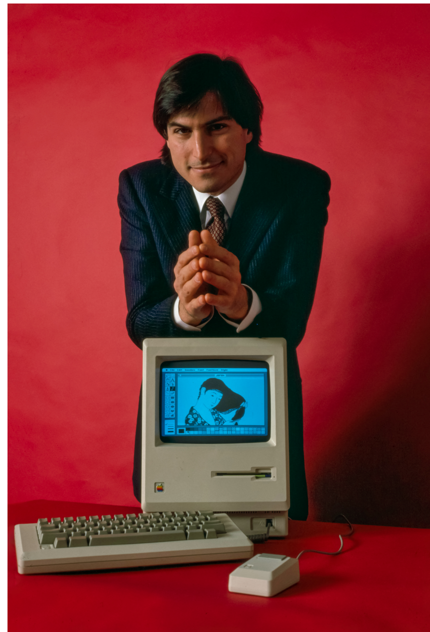
Advisor:
Philippe BONNET

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in

Data-intensive Systems and Applications
Computer Science

August 12, 2024



"Your time is limited, so don't waste it living someone else's life. Don't be trapped by dogma — which is living with the results of other people's thinking. Don't let the noise of others' opinions drown out your own inner voice. And most important, have the courage to follow your heart and intuition. They somehow already know what you truly want to become. Everything else is secondary." [28]

Steven Paul Jobs

Abstract

Delilah: Efficient eBPF Offload for Integrated Data Pipelines on Computational Storage

by Niclas HEDAM

In the two decades since the end of Dennard scaling, improving processing capabilities without increasing processor frequency has become a key challenge for computer science researchers. Concurrently, while storage device throughput has increased exponentially, the throughput between CPU and memory has only improved linearly. Computational storage, which moves data operations closer to their physical storage locations, has gained considerable attraction to address this contrast. This interest has recently reached a critical milestone with standardisation efforts by the Storage Networking Industry Association (SNIA) and Non-Volatile Memory Express (NVMe), proposing eBPF, a vendor-neutral lightweight instruction set architecture for program offload. Simultaneously, integrated data analysis (IDA) pipelines have emerged, combining various programming paradigms, cluster resource management systems, data formats, and execution strategies into unified data management frameworks, allowing for more efficient usage of computational storage.

Despite the standardisation efforts in computational storage, the question of effectively utilising eBPF within the storage layer remains open. Few empirical studies evaluate the performance implications and potential benefits of integrating eBPF with computational storage. Lastly, many open questions exist concerning the implementation of a computational storage device running eBPF, including memory management and cache coherency.

In this thesis, we surveyed the current state of computational storage, identifying fundamental limitations, such as the historical lack of standardised interfaces and short-lived, non-stateful memory. To explore these issues, we designed and implemented Delilah, the first public eBPF-based computational storage processor. Through this implementation, we investigated the challenges, opportunities, and performance characteristics of computational storage. Our findings reveal significant issues related to memory management and cache coherence that impact performance. Despite these challenges, when optimised, Delilah demonstrated the potential for improved performance in specific operations, such as filtering.

Resumé

Delilah: Efficient eBPF Offload for Integrated Data Pipelines on Computational Storage

Af Niclas HEDAM

I de to årtier, der er gået siden enden på Dennard-skaleringen, er det blevet en vigtig opgave for datalogiske forskere at forbedre processorkapaciteten uden at øge processorfrekvensen. Samtidig er kapaciteten i lagerenhederne steget eksponentielt, mens kapaciteten af overførslerne mellem CPU og hukommelse kun er øget lineært. Computational storage, som flytter dataoperationer tættere på deres fysiske lagerplaceringer, har modtaget betydelig interesse for at løse denne udfordring. Denne interesse har for nylig nået en kritisk milepæl med standardiseringsarbejdet fra Storage Networking Industry Association (SNIA) og Non-Volatile Memory Express (NVMe), der foreslår eBPF, en leverandørneutral letvægts-instruktionssæt-arkitektur til program-offload. Samtidig er der udviklet pipelines til integreret dataanalyse (IDA), som kombinerer forskellige programmeringsparadigmer, ressourcehåndteringssystemer til clusters, varierende dataformater og eksekveringsstrategier i samlede datastyringssystemer, hvilket giver mulighed for at udnytte computational storage mere effektivt.

På trods af standardiseringsindsatsen inden for computational storage er spørgsmålet om effektiv udnyttelse af eBPF i storage-laget stadig ikke besvaret. Der er kun få empiriske undersøgelser, der evaluerer konsekvenserne for ydeevnen og de potentielle fordele ved at integrere eBPF med datalagring. Endelig er der mange uafklarede spørgsmål vedrørende implementering af en computerlagerenhed, der kører eBPF, herunder hukommelsesstyring og cache-kohærens.

I denne afhandling undersøgte vi den nuværende situation for computational storage og identificerede grundlæggende begrænsninger, som f.eks. den historiske mangel på standardiserede grænseflader og kortvarig, ikke-tilstandsbestemt hukommelse. For at udforske disse problemer designede og implementerede vi Delilah, den første offentligt tilgængelige eBPF-baserede processor til datalagring. Gennem denne implementering undersøgte vi datalagringens udfordringer, muligheder og egenskaber. Vores resultater afslører betydelige udfordringer i forbindelse med hukommelsesstyring og cache-kohærens, som påvirker ydeevnen. På trods af disse udfordringer demonstrerede Delilah, når den blev optimeret, potentialet for forbedret ydeevne i specifikke operationer, såsom filtrering.

Acknowledgements

Pursuing a PhD has been an experience far different from what I initially envisioned. While society often views PhD holders as exceptional intellects capable of achieving anything, I have learned that the true challenge lies in maintaining hope and determination. This journey has tested my strength and endurance through, at times, disappointing research outcomes, long hours dedicated to obscure and undocumented topics, and moments of being let down. However, it has also been full of unforgettable experiences, such as presenting my research to large audiences, travelling to various parts of the world, and being the first to contribute results and conclusions to my research field.

The challenges were, at times, much more significant than what I could handle alone. Therefore, I extend my heartfelt gratitude to the people who have been essential in helping me navigate this journey. Their tireless support, guidance, and encouragement have made all the difference, and I am deeply thankful for their presence in my life.

Henna Ranta, my girlfriend, whom I met during my PhD stay abroad in Dresden. You came into my life when I least expected it, but exactly when I needed you to. Since then, you have cheered and lifted me up when my PhD journey let me down, motivating me to keep going. Your continuous belief in me has inspired me to strive for my best. I could not have completed this without your love, patience, and support. Kiitos, että olet sinä ja että olet aina uskonut minuun. Minä rakastan sinua.

Thanks to **my family**, who always told me how proud they are and have already shown a lifetime's love. I am thankful for every one of you, particularly my mother, **Sanne**, who has always been there, day or night, to listen on bad days and cheer me up, as well as my grandparents, **Tove** and **Ole**, who always ensured my apartment was stocked with everything I needed, freeing up my energy to focus on research.

Philippe Bonnet, my PhD advisor, for guiding me from my bachelor's to PhD. This summer, we will have worked together for no less than 5.5 years. I could not have done it without you, and I am happy for all the knowledge you shared with me over the years and the people you have introduced me to.

Alex Krause for taking good care of me while I was staying in Dresden. Knowing that a friend was always a few steps down the corridor when I needed one was comforting. Thanks for all the times you invited me out for drinks and all the memes you sent me. Thanks to **Ulrike Schöbel** for always helping with everything in Dresden, from administrative questions to finding out where I could get the cake for the famous *KuK* tradition. And, of course, a heartfelt thank you to the whole team in Dresden for having me and taking such good care of me during my months there.

Búgvi Benjamin Magnussen, **Nikolaj Bläser**, **Andreas Blanke**, **Jakob Mollerup**, and **Magnus Krøyer** for being there every day for five years during my bachelor's and master's studies. It has been fantastic to have a group of friends with whom to celebrate the highs and survive the lows. Thanks for pushing me to improve at what I do, whether that is English grammar, programming in obscure languages, or handling a wide range of academic challenges.

Analog, my little Heaven on Earth. I cannot thank all of you individually, but I am grateful that I had a place to retreat when research and work became too much. Thanks, everyone, for always being there for me, cheering me up on rough days and ensuring that I could always get long and happy conversations or a heartfelt hug when needed.

Thanks to all my colleagues in the **Data-intensive Systems and Applications** group. I am grateful for your support and the advice given throughout my PhD.

Julie Jacobsen, **Vibe Mathiasen**, **Jeppe Hedal**, **Rikke Dandanell**, and **Camilla Frandsen** from PhD Support. Thank you for supporting me when the system dropped or forgot me. Thanks for all our entertaining and spontaneous conversations in the PhD Support offices.

Thanks to the small Finnish village of **Peräseinäjoki** and all the Finnish people for welcoming me and providing a writing refuge in the final months.

And to **all the other people** who somehow became part of this wild journey: Thank you!

Contents

Abstract ii

Resumé iii

Acknowledgements iv

1 Introduction 1
- 1.1 Context 1
- 1.2 Problem 2
- 1.3 Approach 3
- 1.4 Contributions 3
- 1.5 Structure of the Manuscript 5

I State of the Art 7

2 eBPF 8
- 2.1 Structure 9
- 2.2 Execution Environments 10
  - 2.2.1 hBPF 10
  - 2.2.2 uBPF 10
  - 2.2.3 rBPF 11
- 2.3 Alternatives 11
- 2.4 Conclusion 12

3 Program offload 13
- 3.1 Data shipping vs Function shipping 13
- 3.2 Network 14
- 3.3 Storage 14
  - 3.3.1 Willow 15
  - 3.3.2 Biscuit 15
  - 3.3.3 YourSQL 16
  - 3.3.4 OX 16
  - 3.3.5 BlueDBM 16
  - 3.3.6 INSIDER 17
  - 3.3.7 REGISTOR 17
  - 3.3.8 POLARDB 17
  - 3.3.9 NASCENT 17
  - 3.3.10 Eid-Hermes 18

3.3.11 Discussion 19
3.4 Standardisation of Computational Storage 20
3.4.1 Storage Networking Industry Association 20
Theory of Operation 22
Memory Management 23
Program Management 23
Limitations 23
3.4.2 Non-Volatile Memory Express 23
Theory of Operation 24
Memory Management 24
Program Management 25
3.5 Conclusion 25

4 OpenSSD Platform 28
4.1 Interface 29
4.1.1 PCIe 29
4.1.2 Ethernet 30
4.1.3 QSFP28 30
4.2 Zynq UltraScale+ MPSoC 30
4.3 Software Development Kit (SDK) 31
4.3.1 IPs 32
AXI 32
PCIe DMA/Bridge – XDMA 32
MIG 33
GPIO 33
4.3.2 Petalinux 34
4.3.3 Vivado 35
4.3.4 Vivado HLS 36
4.4 UDMA 37
4.5 Conclusion 38

5 Data Pipelines 40
5.1 DuckDB 40
5.2 TensorFlow 41
5.3 PyTorch 41
5.4 Spark 42
5.5 Flink 42
5.6 Dask 43
5.7 DAPHNE 43
5.8 Conclusion 45

6 Summary 47

II Contributions: Delilah 51

7 Requirements 52

8 Design **55**
8.1 Design Decision 1: Memory Management 55
8.2 Design Decision 2: uBPF 57
8.3 Design Decision 3: eBPF Sidestepping 57
8.4 Design Decision 4: Daisy OpenSSD 58
8.5 Design Decision 5: Extend Eid-Hermes 58
8.5.1 io_uring 58
8.5.2 Shared Data Slot 59
8.6 Summary 60

9 Implementation **61**
9.1 Board Configuration 61
9.1.1 Delilah Controller 61
9.1.2 Device-tree 63
9.1.3 ECC initialisation 64
9.2 Protocol 64
9.2.1 64-bit support 64
9.2.2 ehcmddone race-condition 65
9.2.3 Interrupt masking 65
9.3 Driver 65
9.4 Device Controller 68
9.4.1 Modules 68
Command 68
Config 69
Functions 69
Hermes 70
Hardware 70
Interrupt 71
Loader 71
Memory 73
Utilities 75
TSL 76
9.5 Selective Cache Invalidation 76
9.6 Block Design 78
9.6.1 Lessons Learnt 79
9.7 Hardware-accelerated Filtering 80
9.7.1 Lessons Learnt 82
9.8 Conclusion 83

10 Summary **85**

III Evaluation **87**

11 Memory Access Evaluation **88**
11.1 DMA vs. CMB 88
11.2 Memory Accesses 90
11.3 Conclusion 94

12 Experimental Evaluation **95**
12.1 Experimental Setup . . . 96
12.2 Experiment 1: Interpretation vs. Just-in-Time . . . 96
12.3 Experiment 2: Partitioned Execution . . . 97
12.4 Experiment 3: Filesystem Caching . . . 99
12.5 Experiment 4: Filtering Strategies . . . 101
12.6 Experiment 5: Hardware Acceleration . . . 103
12.7 Experiment 6: Partial Offload . . . 103
12.8 Experiment 7: Slot Placement . . . 105
12.9 Discussion . . . 106

13 Lessons Learnt **109**
13.1 Memory Management . . . 109
13.1.1 Delilah and NVMe . . . 111
13.2 eBPF Complements Computational Storage . . . 112
13.3 Workload-Awareness Is Key . . . 113

14 Summary **116**

15 Conclusion **118**

Glossary **120**

Bibliography **127**

List of Figures

3.1 A diagram showing the relationship between the SNIA computational storage terminology. 21

4.1 The Daisy OpenSSD [15]. 29
4.2 A diagram showing the physical components of the Daisy OpenSSD [15]. 30
4.3 An overview of the steps of a single DMA transfer. 31
4.4 An overview of the architecture of the UDMA driver and how it interacts with the kernel [26]. 38

5.1 An overview of the architecture of DAPHNE, which is divided into four architectural tiers [16]. 44

9.1 Overview of the proposed Delilah architecture [23]. The blocks coloured light green denote components belonging to Delilah, and blocks in grey denote components from other vendors. . . 62

11.1 Average latency of a single memory access when running 32 million accesses in a row. The results are shown in linear scaling. *Seq* denotes sequential, *rand* denotes random, and the values in parenthesis denotes the alignment. 91
11.2 Average latency of a single memory access when running 32 million accesses in a row. The results are shown in logarithmic scaling. *Seq* denotes sequential, *rand* denotes random, and the values in parenthesis denotes the alignment. 92

12.1 The result of a full offload of query 3.3, where we compare the runtime between running Delilah in JIT mode versus interpreted mode. In JIT mode, all query instructions are transpiled into ARM instructions native to the Daisy OpenSSD. In interpreted mode, an interpreter parses the instructions and executes them sequentially. 97
12.2 The result of the host-only experiment, where Delilah only fetches the files for the host without doing any processing. We compare the runtime between running Delilah in JIT mode versus interpreted mode. In JIT mode, all query instructions are transpiled into ARM instructions native to the Daisy OpenSSD. In interpreted mode, an interpreter parses the instructions and executes them sequentially. 98

12.3 The result of running the full query on Delilah, either running to completion or in chunks of 128 MB. 99
12.4 The result of running the full query on Delilah, either reading column files with or without the *O_DIRECT* flag. Contrary to other experiments, this experiment is executed with the shared memory slot in PS memory. 100
12.5 The result of running the full query on Delilah with various approaches to filtering. 101
12.6 The result of running the full query on Delilah with hardware-accelerated filtering on the *Filter City* operation. 102
12.7 Three different degrees of offloading the SSB 3.3 query, spanning from full offload to partial offload with aggregation on host and host-only. 104
12.8 Two previous experiments, the baseline offloaded and ARM Neon implementations, with the shared data slot in PL and PS memory. 105

13.1 The traditional understanding of memory hierarchy, where the upper levels are always subsets of the levels below. 109
13.2 The new understanding of memory hierarchy with computational storage, where multiple memory hierarchies share a lower layer but have separate upper layers. This complicates moving data in and out of various levels of caches across a collection of compute units. 110

List of Tables

2.1 A select few eBPF instructions, their mnemonic meaning, pseudo-code, and a simple description. 9

3.1 An overview of the command register of Eid-Hermes. 19
3.2 An overview of the command control register of Eid-Hermes. 19

9.1 The distinct states of Eid-Hermes/Delilah engines. 65
9.2 The memory latency of accessing PS and PL, respectively. . . . 76
9.3 The runtime of offloading a program to Delilah with SCI disabled that reads a file from the underlying SSD and transfers it to host memory. 77
9.4 The runtime of offloading a program to Delilah with SCI enabled that reads a file from the underlying SSD and transfers it to host memory. 77

11.1 The latency of writing various amounts of data from the host to data slots via Controller Memory Buffers (CMB). 88
11.2 The latency of writing various amounts of data from the host to data slots via Direct Memory Access (DMA). 89
11.3 The latency of writing various amounts of data from the host to data slots via Direct Memory Access (DMA) using all four H2C/C2H channels. 89

12.1 Design principles and assumptions derived from experimenting with Delilah on the Daisy OpenSSD. 108

List of Listings

4.1 An example of an HLS function with parameters and the correpsonding pragmas. 37
9.1 The contents of .profile 62
9.2 Declaration of the region reserved for the BAR0 register. . . . 63
9.3 Declaration of the consumer of the region reserved for the BAR0 register. 63
9.4 The signature of Command handlers in Delilah. 68
9.5 The simplest implementation of a command handler in Delilah. The Clear State command is in charge of setting the internal shared data slot to 0s. 69
9.6 The exposed functions of the IRQ module to be implemented. 71
9.7 The exposed functions of the Loader module to be implemented. 71
9.8 The implementation of the worker thread. 72
9.9 The exposed functions of the Memory module to be implemented. 73
9.10 The signature of our hardware accelerated filtering operator. . 80
9.11 The implementation of one of the operations of our hardware accelerated filtering operator. 81
12.1 Star Schema Benchmark Query 3.3. 95

Chapter 1

Introduction

1.1 Context

In recent years, the rise of artificial intelligence, machine learning, and large-scale data analytics has been fueled by the availability of increasingly large volumes of data. While storage and network capacities have increased exponentially, processing capacity has only seen linear growth [47]. Consequently, data volumes (measured in TB) are growing much faster than processing capacity (measured in Gops/sec). One solution of addressing this imbalance is to attach processing capabilities directly to storage devices, enabling computations directly on stored data, independently of the host processing unit. This concept is known as computational storage [33].

Computational storage enables a host or remote processing unit to access SSDs through a computational storage processor (CSP), an embedded processor on a PCIe endpoint. This processor can run fixed programs or be programmable, allowing host applications to offload parts of their internal logic to the computational storage processor.

The Storage Networking Industry Association (SNIA) has proposed several mechanisms for code offload onto computational storage [9, 5]:

- **Bitstream-based offloading for FPGAs**: This involves synthesising specific or generic hardware accelerators and placing them on the board. This method is static and requires reprogramming and rebooting to offload new procedures.
- **Operating System-based offloading**: The storage controller is bundled with an operating system, often Embedded Linux. For Xilinx devices, for example, the OS is generated by the Petalinux SDK. This method is practically static, as changes require recreating and redeploying the OS.
- **Container-based offloading**: Computational storage capabilities are bundled in containers, allowing easy modification or replacement.
- **eBPF-based offloading**: The computational storage controller can execute programs shipped from the host and represented in eBPF byte-code. This method is the most dynamic, enabling the offloading of

entire programs, looping routines, and functions with multiple paths depending on run-time information.

BPF, defined in 1992 as an efficient network packet filtering system, allowed filtering of incoming network packets with 20 times the efficiency of the state-of-the-art at the time [21, 5]. In 2014, BPF was redesigned as eBPF, focusing on modern hardware. eBPF is compiled from a high-level language, often C, into eBPF bytecode. This bytecode is interpreted by a VM or JIT compiled to native instructions. eBPF VMs and JIT compilers are defined for various processors, including x86_64, ARMv8, and RISC-V. In computational storage, eBPF can be utilised as a vendor-neutral ISA, allowing application developers to write programs across different devices. The role of eBPF in storage devices remains to be fully explored.

Simultaneously, integrated data pipelines are emerging, unifying domains such as data management (DM), query processing, high-performance computing (HPC), and machine learning (ML) training and scoring. This unification paves the way for a new class of computational storage devices where data persists longer than a single operator. Such devices must have systems that allow the host to manage device-side memory efficiently. With eBPF being a general-purpose ISA, it is promising as an offload mechanism for this new class of devices.

In addition, with recent significant advancements in the OpenSSD project, we now have the hardware necessary to explore this new class of devices.

1.2 Problem

Several questions need to be addressed with the emergence of eBPF as the industry standard for code offload on computational storage.

Firstly, how can we efficiently leverage eBPF code offload within a data pipeline? We must understand how the architecture of an eBPF execution environment on computational storage may look. Furthermore, what does it mean to utilise eBPF to offload processing tasks to a computational storage processor? What are the advantages and limitations of this approach? Finally, what are the broader implications for overall system design?

To answer these questions, we aim to design and implement a computational storage processor (CSP) that utilises eBPF as its offload mechanism. Before doing so, we must understand the current landscape of computational storage and examine previous proposals for computational storage devices. Additionally, we investigate the state-of-the-art data pipeline technologies to understand the use cases of computational storage. Furthermore, we explore the differences between traditional data pipelines and integrated pipelines.

Our experimental approach includes building a computational storage device based on the Eid-Hermes project [40] and deploying it to the Daisy

OpenSSD [13]. This novel device will allow us to evaluate the practical application of eBPF as an offload mechanism and understand the challenges of dynamic code-offload, including memory management and cache coherency.

Thus, the central question we address in this thesis is:

How can we efficiently offload eBPF code from a host integrated data pipeline to computational storage?

Lastly, while efficiency can be interpreted in many different ways, we define *efficiently* as improving the overall performance of a given set of operations. For example, we consider eBPF code offload efficient if the offload results in higher throughput or lower execution time.

1.3 Approach

We approach this problem with an experimental mindset, acknowledging that we have limited information on the consequences, opportunities, and limitations of utilising eBPF in the context of storage. Specifically, our understanding of how eBPF and the OpenSSD boards perform when offloading gigabyte-scale operations is minimal. One critical question is whether the clock frequency of these devices is sufficient to handle such high-volume workloads efficiently and whether or not eBPF is too generic to represent the complexity of storage operators.

To explore these uncertainties, we begin by establishing an experimental setup. This setup includes a host and a monitoring/development machine connected to a Daisy OpenSSD. The host machine is linked via PCIe, while the development machine is connected through JTAG. Our initial step involves deploying one of the example block designs provided for the device. Concurrently, we start developing a simple controller within the processing system. Initially, this controller will be compatible with Eid-Hermes but will be extended with additional functionalities as our work progresses.

As the controller and its corresponding driver become sophisticated enough for experiments, we reach a point where we can conduct meaningful experiments to evaluate the implications of offloading eBPF to this new class of devices. These experiments will focus on the devices' performance characteristics, exploring how well they manage gigabyte-scale workloads, whether the clock frequencies and other hardware specifications are adequate for such tasks, and whether eBPF can express storage operations efficiently.

By adopting this experimental approach, we aim to gain new knowledge of eBPF's potential within storage systems, ultimately laying the groundwork for developing more efficient and flexible storage solutions.

1.4 Contributions

Our contribution to computational storage is threefold:

Firstly, we explore the current landscape of computational storage. This exploration includes examining current efforts to standardise the vendor-neutral offload and reviewing past and present attempts at building computational storage devices. We analyse and discuss different architectures and design choices' opportunities, challenges, and limitations. Additionally, we study modern data pipelines to understand which use cases and interfaces can benefit the most from computational storage. We also closely examine eBPF as a possible method for offloading tasks to computational storage.

Secondly, based on the findings of our state-of-the-art review, we propose a new software and hardware architecture. This design is implemented on the Daisy OpenSSD platform, allowing us to evaluate the design decisions, including protocol and memory management. Our implementation uses eBPF to help host data pipelines offload dynamic operations and manage memory more efficiently.

Our implementation is available on GitHub across several repositories:

- `delilah-csp/delilah`: A flexible and lightweight computational storage controller based on eBPF, written in C. The controller is deployed as a Petalinux application and runs in user-space.
- `delilah-csp/delilah-pt`: Petalinux configurations and patches for deploying the Delilah controller on the Daisy OpenSSD board. These patches and configurations include, for example, memory initialisation and device-tree specifications
- `delilah-csp/delilah-bd`: Block-design files for the Daisy OpenSSD, including all necessary IPs for deploying Delilah. This block design includes, for example, IPs to connect to the host over PCIe and memory management IPs.
- `delilah-csp/delilah-host`: A host-side driver that exposes Delilah's functionality via io_uring. The driver utilises XDMA by Xilinx/AMD and is based on the Eid-Hermes driver and protocol [40].
- `delilah-csp/delilah-hw-filter`: A high-level synthesis hardware accelerator for filtering large datasets within Delilah, written in C.

Thirdly, we design and conduct several experiments to learn from our proposed architecture and its components. These experiments provide concrete insights into contemporary computational storage design and how devices should be integrated into modern data pipelines. We provide practical lessons on improving data processing efficiency, scalability, and overall system performance with computational storage.

In summary, the thesis lays the foundation for further exploration and design of computational storage. We both provide the theoretical understanding and show the practical application of computational storage.

The thesis is partially based on a publication at the ADMS workshop and three DAPHNE deliverables:

- N. Hedam, M. T. Clausen, P. Bonnet, S. Lee, and K. F. Larsen. 2023. Delilah: eBPF-offload on Computational Storage. In *Proceedings of the 19th International Workshop on Data Management on New Hardware* (DaMoN '23). Association for Computing Machinery, New York, NY, USA, 70–76 [23].
- D6.3 Prototype and Overview of Data Path Optimizations and Placement [42].
- D6.2 Prototype and Overview of Managed Storage Tiers and Near-Data Processing [6].
- D6.1 Computational Storage Capabilities [5].

Lastly, some of the experimental analyses of Delilah using the Star Schema Benchmark (SSB) were conducted in collaboration with Alexander Krause and Johannes Pietrzyk from the Technische Universität Dresden. The work is currently unpublished.

1.5 Structure of the Manuscript

This manuscript is divided into three essential parts. Part I is dedicated to state-of-the-art computational storage; Part II is dedicated to our contribution to computational storage; and Part III is dedicated to evaluating our contribution. Before the three parts, Chapter 1 introduces the thesis and research domain.

In Part I, we explore the state-of-the-art of computational storage. Chapter 2 describes and discusses eBPF, an emerging instruction set architecture, suitable for offloading to computational storage. Chapter 3 explores the landscape of program offload to computational storage, including contemporary standardisation efforts and previous attempts at designing computational storage devices. Chapter 4 describes the OpenSSD platform and how it can be used to experiment with and evaluate a computational storage device. Chapter 5 explores the characteristics of several contemporary systems often used for designing and implementing data pipelines. Chapter 6 summarises the findings and the lessons learnt in this part.

In Part II, we present our primary contribution: a computational storage processor running eBPF and exposing host-manageable memory spanning several operators. Chapter 7 discusses the requirements for a novel computational storage processor. The chapter is based on our findings in Part I. Chapter 8 outlines our contribution's concrete design, including arguments and necessary changes to previous work. Chapter 9 describes in depth the implementation details of our contribution. Chapter 10 summarises the requirements and design, and how they correspond to the concrete implementation of our computational storage processor.

In Part III, we evaluate our contributions. Chapter 11 provides a set of architectural experiments. These architectural experiments show the performance

of the OpenSSD device and how efficient its different components are. Chapter 12 evaluates the Delilah integrated into a host-side application, emulating the behaviour of the SSB query 3.3. Chapter 13 provides a list of lessons learnt from building and experimenting with Delilah. Chapter 14 summarises the experiments and the lessons learnt.

Finally, Chapter 15 concludes the manuscript with a summary of the state-of-the-art, our contributions, evaluations, and any future work.

We provide a glossary at the end of the thesis to give an overview of domain-specific terms and abbreviations, including the pages on which they are mentioned.

Part I

State of the Art

Chapter 2

eBPF

Any environment that can execute remote or offloadable instructions must have a structure to represent programs. In the context of networking and kernel probes, BPF and eBPF have been around for decades, serving precisely this purpose.

The Berkeley Packet Filter (BPF), introduced in 1992, enabled user-space programs to execute functions within the Linux kernel without using kernel modules. The original version of BPF was, as the name indicates, targeted at network packet filtering. It defined a bytecode structure alongside a virtual machine (VM) embedded in the Linux kernel. In 2014, Alexei Starovoitov introduced Extended BPF (eBPF) as a modern adaptation of BPF designed for contemporary processors. Unlike BPF, there is currently no official standardisation body for eBPF, although eBPF Foundation serves to maintain the technical direction and vision of eBPF. Consequently, the latest version of the eBPF bytecode is the one that the Linux kernel can interpret. In-kernel eBPF JIT compilers are available for various architectures, including x86, ppc64, s390x, mips64, sparc64, and ARM. Due to the simplicity of eBPF, most eBPF instructions can be directly mapped onto native instructions of these architectures.

Both GCC and LLVM offer eBPF backends, enabling them to generate eBPF code from C programs. In the context of user-space execution, the IOVisor project introduced a VM called User-space BPF (uBPF), which we will cover in section 2.2.2. uBPF is an Apache-licensed library built for executing eBPF programs [57]. It is released under an Apache license on GitHub. Unlike the Linux kernel implementation, which is under the GPL license, uBPF provides a user-friendly alternative for executing eBPF programs in user-space.

As such, the critical strength of eBPF is the well-defined and vendor-neutral nature of the instruction set, combined with several Apache-licensed execution environments. These characteristics make eBPF particularly attractive, given its ease of integration and compatibility across different platforms. Additionally, the open-source nature of its execution environments invites the community to contribute to further development.

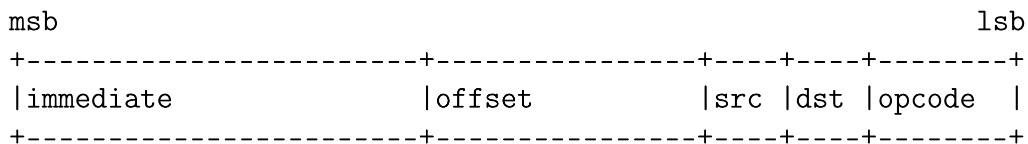
However, eBPF also has a critical weakness. Since eBPF is a lightweight, vendor-neutral, general-purpose instruction set architecture, the ability to

write specialised programs in eBPF is severely limited. This restriction can result in simple programs being compiled into numerous instructions, thus limiting performance artificially. Additionally, the data structures that eBPF supports are limited, as it lacks instructions for floating-point arithmetic and dynamic memory allocation.

This chapter explores eBPF as a potential instruction set architecture (ISA) for program offload. We want to understand what eBPF can do while considering any limitations to the architecture and execution environments.

2.1 Structure

eBPF programs consist of a sequence of 64-bit instructions. These instructions are organised into an immediate, offset, source, destination, and opcode, going from most significant to least significant bit [32]. The immediate field holds any constants given to instructions, while the offset field indicates an offset, often in the context of a jump. Source and destination indicate which registers load input or store output values. Lastly, opcode holds the operation code itself. It should be noted that instructions often use only a subset of the fields.



Opcode	Mnemonic	Pseudocode	Description
0*x*07	add dst, imm	dst += imm	Increase *dst* by *imm*
0*x*67	lsh dst, imm	dst ⟨⟨= imm	Bitshift *dst imm* times
0*xb*7	mov dst, imm	dst = imm	Set *dst* to *imm* (int64)
0*x*18	lddw dst, imm	dst = imm	Set *dst* to *imm* (word)
0*x*05	ja +off	PC += off	Jump *imm* instructions ahead
0*x*6*d*	jsgt dst, src, +off	PC += off if dst ⟩ src	Jump *imm* instructions ahead, if *dst* ⟩ *src*
0*x*85	call imm	call imm	Call external function no. *imm*
0*x*95	exit	return r0	Return *imm*

TABLE 2.1: A select few eBPF instructions, their mnemonic meaning, pseudo-code, and a simple description.

As of 2017, eBPF supports 102 different instructions [32]. We show eight examples of eBPF instructions in table 2.1.

2.2 Execution Environments

2.2.1 hBPF

hBPF is an experimental project aimed at enabling eBPF program execution on hardware [49]. Initially released in early 2021, hBPF explores the feasibility and performance of alternative hardware description languages and cost-effective development boards like the Arty-S7. This approach opposes traditional proprietary toolchains and expensive multi-core accelerator cards like the Xilinx Alveo. Its primary goal is to provide insights and support experimentation rather than competing in high-performance production environments.

While hBPF remains under active development, it has certain limitations compared to in-kernel eBPF. Notably, hBPF has yet to be fully optimised and does not support stack utilisation. Due to this, the R1 register is repurposed for input arguments instead of serving as a base pointer. Furthermore, hBPF extends the CPU's functionality through call handlers using the call opcode.

However, despite the attention and promising functionality, hBPF's immaturity renders it unsuitable for high-performance or production environments in its current state.

2.2.2 uBPF

uBPF is a user-space execution environment for eBPF, aiming to provide an Apache-licensed library for executing eBPF programs. This characteristic contrasts with the kernel's implementation, released under the GPL license, limiting its usage in many projects. By offering a user-space alternative with an Apache license, uBPF intends to provide greater accessibility for non-GPL compatible projects seeking to incorporate eBPF functionality.

The uBPF VM is a RISC register machine with eleven 64-bit registers, including one stack pointer, an implicit program counter, and a fixed-size stack. It provides access to several registered functions, typically used for BPF helpers, although it can handle any registered function behind the scenes. The uBPF VM can accept either a buffer containing eBPF instructions or an eBPF ELF file. Its loader parses the segment header table and sections to extract the program and references to registered functions.

The uBPF VM offers a simple interface of three operations: loading a program, unloading/resetting the uBPF engine, and executing the loaded program with a provided memory buffer.

When an eBPF program executes within the uBPF VM, it must return a value by storing it in register 0 of the VM. This return value may be a scalar value, an integer representing a pointer to memory managed by the uBPF VM, or an identifier for a resource managed outside the uBPF VM.

2.2.3 rBPF

rBPF, developed by Quintin Monnet, is a Rust-based execution environment for eBPF, originating as a port of uBPF to Rust [36]. Despite their similarities, there are several differences between uBPF and rBPF:

1. **Constants**: Some constants, such as the maximum program length or stack length, differ between uBPF and rBPF. rBPF aligns with the Linux kernel's values, while uBPF maintains its own.
2. **Error Handling**: When errors occur during eBPF program execution, uBPF quietly returns the maximum value as an error code, whereas rBPF returns Rust's `Error` type.
3. **Registered Functions**: The registration of helper functions, callable from within an eBPF program, is handled differently between the two environments. In rBPF, any function can be registered, whereas the `call` instruction in uBPF has limitations.
4. **Performance**: Theoretically, Just-in-Time (JIT) compiled programs are expected to run at similar speeds to uBPF. However, uBPF's C interpreter may outperform rBPF's interpreter, though this has yet to be benchmarked.

It is worth noting that rBPF, like hBPF, is under active development, and its API may undergo changes. Additionally, while most eBPF instructions are implemented, there are some unsupported instructions. As with hBPF, the immaturity of rBPF renders it unsuitable for high-performance or production environments in its current state.

2.3 Alternatives

For the offload to storage or networking, we use eBPF as a mechanism to structure a sequence of instructions in a vendor-neutral manner. As such, any vendor-neutral instruction set architecture may be used for offload.

However, studies have previously explored instruction set architectures suitable for program offload, focusing on computational storage. Huang and Paradies explored the characteristics of WebAssembly and compared them to those of eBPF [25]. WebAssembly was initially developed for secure and efficient execution in web browsers. However, the authors state that recent research has shown that WebAssembly still needs work in terms of security and efficiency.

The study concluded that WebAssembly is superior to eBPF for program offload to computational storage. However, the study also shows superior startup and preparation time for eBPF execution and similar execution time between eBPF and WebAssembly. While the authors acknowledge that eBPF overperforms WebAssembly, they believe that the restrictions and limitations make eBPF unsuitable for production.

2.4 Conclusion

eBPF continues the original BPF (Berkeley Packet Filter), introduced in 1992. eBPF enables user-space programs to execute within the Linux kernel without requiring kernel modules. eBPF was designed for contemporary processors by Alexei Starovoitov in 2014, with its direction overseen by the eBPF Foundation.

eBPF programs consist of 64-bit instructions with `immediate`, `offset`, `source`, `destination`, and `opcode` fields.

eBPF has emerged as a candidate for a general-purpose instruction set architecture in various domains. One of its key strengths is vendor neutrality, making it a strong candidate for offloading to other domains.

By being vendor-neutral, eBPF allows for the development of storage applications not tied to specific devices, decreasing costs and increasing interoperability.

However, although eBPF is a promising candidate, Huang and Paradies conclude that WebAssembly may be superior to eBPF as a general-purpose instruction set architecture for offload to storage. However, their study also found that eBPF typically performs better than WebAssembly. Thus, which instruction set architecture is best suited for offloading to a particular domain depends on context.

Chapter 3

Program offload

The concept of computational storage and offloading to a device or server is not new. Computational storage has gained significant traction after Dennard scaling ended in the mid-2000s. Since then, there have been several attempts at designing and implementing architectures to increase processing capabilities and throughput in high-performance systems. In this chapter, we discuss the strengths and weaknesses of such architectures.

3.1 Data shipping vs Function shipping

In a publication from 1998, Kossmann and Franklin explored the two different ways queries are executed in client-server database systems [31]. They compared two typical methods: data shipping, which is when queries run on client machines, and function shipping, which is when servers entirely handle query execution. They also studied a hybrid approach that combines parts of both methods. They concluded that this hybrid model often performs the best compared to shipping either data or queries only.

Voruganti et al. conducted a survey in 2004, with similar results to Kossmann and Franklin's conclusions [58].

In the 25 years after Kossmann and Franklin's study, we have seen similar developments in storage. Historically, storage devices were accessed from the host through simple interfaces, where the host acts as a client to the storage device. With the introduction of NVMe, we saw a trend of moving some functionality to the device. For example, Open-Channel SSDs enabled the host to leverage device geometry to improve performance [48]. Another example is OX, further described in Section 3.3.4, which enabled the host to modify the device FTL to match application characteristics.

However, this is where the similarities between Kossmann and Franklin's study and storage end. Storage has the notable difference that I/O requests are part of *function shipping* and complicate the architecture.

The emergence of computational storage and eBPF introduces new and exciting perspectives. To some extent, it is a modern form of function shipping, where queries and pipelines are compiled into eBPF code and shipped to an accelerator or computational storage device. On the other hand, with I/Os

being part of the shipped function, with asynchronous side effects, the findings of Kossmann and Franklin are likely not relevant for functions offloaded to storage.

3.2 Network

Since most network packets are fully standardised through the Internet Protocol version 4 (IPv4) or Internet Protocol version 6 (IPv6), it is straightforward to design mechanisms for program offload, as exemplified by the offload of BPF and eBPF to SmartNICs. The input of an offloaded program is always one or more network packets of a known format, and the output is always a decision on how to handle the packet at hand. Due to the potentially high volume of packets, the number of instructions per packet is also limited. Furthermore, it is always considered appropriate to do nothing and forward packets wherever a particular decision is not apparent.

Due to the above, program offload to network challenges are more trivial than offload to non-heterogeneous components like storage devices. This view is substantiated by related work. For example, Gibb et al. defined design principles for packet parsers in 2013, where they showed how reconfigurable packet parsers should work [19]. Their abstract parser model reads packet header data to determine which header fields are present and extract relevant values to a field buffer. After extraction, a match engine finds the appropriate parser.

Gibb et al. further argue that the need for programmability stems from the fact that header formats may change after the deployment of the initial parser, or if the network operator is utilising custom headers. However, this proves that network packets are standardised enough for parsers to be reprogrammed easily. In a network packet, the header fields are always prepended to the payload, and an understanding of these fields is on a best-effort basis.

3.3 Storage

Contrary to networking, attempts to enable program offload to storage devices are much more complex. In this section, we provide a non-exhaustive list of computational storage architectures to understand better the design choices of previously proposed computational storage devices. We are interested in understanding what class of hardware these devices are built atop and the motivations behind the choice.

Architectures such as Willow, BlueDBM, INSIDER, REGISTOR, POLARDB, and NASCENT are FPGA-based, while Biscuit takes an ARM-based approach. YourSQL, on the other hand, builds on a software-based solution. OX and Eid-Hermes propose architectures for offloading to specialised hardware, namely Open-Channel SSDs and eBPF-based accelerators, respectively. This set of architectures prompts several intriguing questions:

- What are the key advantages of constructing an FPGA-based computational storage device, and how does it compare to an ARM-based computational storage device?
- How can offloading be executed efficiently?
- How are memory regions organised and shared with the host?

Each architecture offers unique insights into the landscape of computational storage, highlighting the diverse approaches and considerations in designing efficient and practical storage solutions.

3.3.1 Willow

Seshadri et al. introduced Willow in 2014. It is a prototype system based on FPGA technology and aimed at enabling programmers to extend SSDs with application-specific functionality while keeping filesystem protections intact [54, 18].

Willow is deployed to an FPGA. In Willow, the FPGA is interfaced with the host system via PCIe and offers programmable functionality through *SSD Apps*. An SSD App consists of a series of generic Remote Procedure Call (RPC) handlers, without storage-specific functions, which can be deployed at individual Storage Processor Units (SPUs).

Challenges associated with Willow include limitations in the complexity and performance of Apps, particularly when concurrent SPU transfers are needed, and constraints on the number of Apps that can run simultaneously. Additionally, the system does not support dynamic memory allocation or task offloading for execution on the SSD.

3.3.2 Biscuit

In 2016, Gu et al. proposed Biscuit, an NDP framework designed to provide general-purpose offload through high-level APIs within two libraries. The two libraries are *libslet* and *libsisc*, and they support the development of data-intensive applications in the form of SSDlets [20, 18]. In Biscuit, an SSDlet is a C++ program based on Biscuit APIs that can be independently scheduled. SSDlets operate via a distributed mechanism on both the host and storage systems, utilising PCIe Gen3 x4 links.

Biscuit uses ARM Cortex-R7 real-time embedded processors located within the SSD firmware and user runtime and hardware pattern matches to offload and execute tasks performantly. The ARM cores were initially used for SSD functions to yield performance advantages. However, Gu et al. failed to demonstrate an increase in performance when running both user applications and host I/O requests concurrently.

Some of Biscuit's key advantages are dynamic module loading and memory allocation. However, despite these software capabilities, Biscuit has several hardware challenges, including cache coherency issues, limited processing

and computing power, synchronisation constraints, and insufficient memory or Memory Management Unit (MMU) capacity.

Lastly, Biscuit is a fixed-function computational storage device, limiting users to specific applications due to their flow-based programming model.

3.3.3 YourSQL

Jo et al. proposed YourSQL in 2016. Their architecture uses commodity NVMe SSDs connected to a server via PCIe Gen3 x4. YourSQL supports device-side filtering, user-programmability, and real-time scanning of tables via a hardware pattern matcher [27, 18].

However, the YourSQL implementation has challenges and limitations. MariaDB was modified to integrate its query planner and storage engine with Biscuit (described in Section 3.3.2). These significant changes can potentially lead to security issues and performance degradation due to diverging changes to MariaDB in the future.

YourSQL has a hardware and software layer. Software-based filtering, however, incurs an overhead and may, therefore, impact overall system performance.

3.3.4 OX

OX, proposed by Picoli in 2019, is an architecture for configuring System-on-Chip (SoC)-based storage controllers atop Open-Channel SSDs [46].

OX differs from other computational storage systems by enabling system administrators to tailor custom application-specific Flash Translation Layers (FTLs) to match the characteristics of the database system. This adaptability optimises system performance and ensures more efficient utilisation of resources, catering specifically to the workload characteristics.

However, Picoli et al. stated in 2020 that Open-Channel SSDs cannot be considered a uniform class of devices due to the complicated device geometry and characteristics [48]. As such, NVMe moved on to standardise ZNS devices instead, leading to the de facto abandonment of OX.

3.3.5 BlueDBM

Jun et al. introduced BlueDBM in 2015 [29, 18]. It is an architecture that enables processing in flash-based storage, focusing on data analytics. BlueDBM, like Willow, is deployed to an FPGA. It implements host and network controllers, flash memory, and in-storage processors.

BlueDBM has several limitations and challenges concerning its filesystem, Remote File Sharing (RFS). RFS maintains mapping information, which enables filesystems to retrieve files from their physical locations on the flash.

However, the inability to use other filesystems than RFS may decrease developers' interest and thus limit adoption.

3.3.6 INSIDER

Ruan et al. introduced INSIDER in 2019 [52, 18]. It is based on an FPGA architecture like Willow. It works as a reconfigurable drive controller and utilises PCIe Gen3 for interconnection to the host. Because it is based on FPGA technology, INSIDER supports several different workloads instead of being limited to one. The architecture of INSIDER is divided into a control plane, responsible for firmware logic, and a data plane with accelerators. This division guarantees the security and integrity of user data by preventing unauthorised program access to the drive.

On the host, INSIDER is based on a virtual file abstraction. On the drive, it exposes a simple and concise interface to abstract away data movement implementation details between system components and the device-side processor. This approach removes the need for programmers to develop and maintain customised APIs, which could be incompatible with existing system interfaces or introduce security issues.

3.3.7 REGISTOR

Pei et al. introduced REGISTOR in 2019 [44, 18]. REGISTOR was designed to improve regular expression searches within storage devices using a deep pipeline structure. REGISTOR has two primary components: a hardware architecture implemented on an FPGA and a software stack.

Experiments and analyses of REGISTOR showed that it achieves high throughput and reduces the I/O bandwidth requirement by up to 97%. Additionally, it reduces CPU utilisation by as much as 82% for regex search in large datasets.

3.3.8 POLARDB

Cao et al. introduced POLARDB in 2020. It is a cloud-native Online Transaction Processing (OLTP) database designed by Alibaba Cloud [7, 18]. It uses heterogeneous computing within storage nodes to support early predicate evaluation via table scan pushdown.

Their architecture is based on an FPGA that is host-managed and aims to optimise hardware costs by utilising PCIe as the storage interface. One significant challenge is the need to verify results against a complete table scan, as not all potential scan conditions are supported.

3.3.9 NASCENT

Salamat et al. utilised a Samsung SmartSSD to introduce NASCENT in 2021. It is a near-storage accelerator for database sorting tasks, focusing on bitonic

sort [53, 18]. Their architecture is based on FPGA technology and is designed to maximise parallelism. By connecting the FPGA directly to the system via the PCIe bus, they hoped to optimise performance compared to conventional architectures.

In traditional setups, where a single FPGA is connected via PCIe to storage devices, there is a limit on simultaneous access to multiple SSDs. However, NASCENT supports direct connectivity between a SmartSSD and an FPGA. This configuration enables partition sorting at the SSD level, independent of other SmartSSDs. This significantly improves system performance, especially as the number of storage devices increases.

3.3.10 Eid-Hermes

Eid-Hermes, a project led by Eideticom, is designed to show how eBPF-based accelerators can be used to offload application code from host processors [40, 23]. The Eid-Hermes is divided into four sub-projects: a QEMU Model, an AWS F1 implementation, a Linux driver, and an eBPF userspace library.

Besides the four projects, Eid-Hermes defines a host-controller transport protocol. Eid-Hermes provides the interface for loading, unloading, and executing programs on the device. One of the core principles of Eid-Hermes is the focus on data and program transport. Specifically, Eid-Hermes utilises program slots and data slots as memory buffer abstractions.

During device enumeration, the host system is made aware of the number and sizes of these slots. In the context of computational storage, Eideticom's decision to separate program slots and data slots in the protocol opens multiple doors. For example, it allows for daisy-chaining of programs, where multiple programs use the same data slot. It also decouples the timing of program and data transfer by allowing the host to offload programs before the data is known. The protocol utilises XDMA, a Xilinx DMA engine elaborated in Section 4.3.1, for transferring programs and data to the device.

Eid-Hermes uses three distinct Base Address Registers (BARs). BAR0 contains the command registers and Eid-Hermes state, including device capabilities. BAR2 holds configurations related to XDMA, while BAR4 stores the program and data slots of uBPF. Commands are transmitted and executed by writing to the Eid-Hermes command register on BAR0.

After the driver is installed, it generates a unique device file, `/dev/hermesX`, for each Eid-Hermes device in the system. The interface supports typical system calls, including `open`, `close`, `ioctl`, `write`, and `read`, exposing DMA capabilities and program execution functionality. `ioctl` operations enable program execution with automatic allocation and reuse of program slots. Data transfer is achieved through `write` and `read` system calls, with automatic allocation and reuse of data slots.

Eid-Hermes has certain limitations. Most importantly, each file descriptor is constrained to a single program slot and a single data slot, automatically

Offset	Len	Name	Mode	Description
0x00	4	EHVER	RO	Interface version
0x04	48	EHBLD	RO	Build version (git describe)
0x34	1	EHENG	RO	Number of eBPF engines
0x35	1	EHPSLOT	RO	Number of program slots
0x36	1	EHDSLOT	RO	Number of data slots
0x38	4	EHPSOFF	RO	Base address program slots
0x3C	4	EHPSSZE	RO	Size of a single program slot
0x40	4	EHDSOFF	RO	Base address of data slots
0x44	4	EHDSSZE	RO	Size of a single data slot
0x1000	32	EHCMDREQ 0	RW	Command request for eng. 0
0x1020	16	EHCMDRES 0	RO	Command response for eng. 0
0x10...	32	EHCMDREQ N	RW	Command request for eng. N
0x10...	16	EHCMDRES N	RO	Command response for eng. N
0x2000	8	EHCMDCTRL 0	RW	Command control for eng. 0
0x20...	8	EHCMDCTRL N	RW	Command control for eng. N

TABLE 3.1: An overview of the command register of Eid-Hermes.

Byte	Name	Mode	Description
0	EHCMDEXEC	RW	Host writes `1` to start engine. Cleared by device afterwards.
1	EHCMDDONE	RO	Device writes `1` on completion. Cleared by device before starting, set back to 1 when done.
2-7	–	–	Reserved

TABLE 3.2: An overview of the command control register of Eid-Hermes.

assigned by the driver. Thus, users who need to concurrently execute multiple programs must open `/dev/hermesX` multiple times. Furthermore, if the intention is to pass the resulting output from one program to the input of another program, programs must be consolidated into a single executable to achieve this.

3.3.11 Discussion

Looking across the computational storage landscape, we observed several fundamental tendencies. Most notably, FPGA-based architectures are a popular choice due to their ability to perform fast processing while maintaining low power consumption. In addition, FPGAs offers configurable and flexible architectures. This trend, in turn, allows for implementing customisable data paths focusing on particular metrics or requirements like throughput, power consumption, or latency. On the other hand, processor-based architectures are also popular due to their general-purpose flexibility and ability to run various programs out of the box.

Additionally, some of the architectures focus on more specific use cases. For instance, YourSQL only supports SQL-based offloading, while OX focuses on modifying the Flash Translation Layer (FTL) to match application characteristics.

In conclusion, the design of a successful computational storage architecture may integrate elements from each subset. For example, FPGA hardware could be utilised to define data paths with high throughput, while processors enable general-purpose program execution. It is now possible to design these hybrid architectures due to the emergence of Xilinx UltraScale+ MPSoCs, as discussed in Section 4.2. Such architectures might incorporate static and bounded components, such as parsers, filters, or FTL adaptation support systems, rather than relying solely on one of them as the only component.

3.4 Standardisation of Computational Storage

This section discusses standardisation efforts within computational storage. In the context of storage, Storage Networking Industry Association (SNIA) and Non-Volatile Memory Express (NVMe) are the most well-known contemporary standardisation parties. Both have recently led efforts to standardise program offload to computational storage.

3.4.1 Storage Networking Industry Association

In the early 2020s, the Storage Networking Industry Association (SNIA) defined a specification for program offload to computational storage [10, 9]. Their specification is focused on terminology and relationships between different types of resources, rather than implementation details. They define the term *computational storage* as "architectures that provide computational storage functions coupled to storage, offloading host processing or reducing data movement". To support this definition, they introduce the following new terminology.

- **Computational storage array (CSA)**: A collection of computational storage devices, control software, and optional storage devices. A computational storage array provides computational capabilities to potentially diverse devices.
- **Computational storage drive (CSD)**: A storage element that provides computational storage functions and persistent data storage. The main difference between computational storage arrays and computational storage drives is the number of underlying persistent storage mediums.
- **Computational storage engine (CSE)**: A component that is able to execute one or more computational storage functions. A computational storage engine is a collection of execution environments and device-specific functions compatible with these environments.

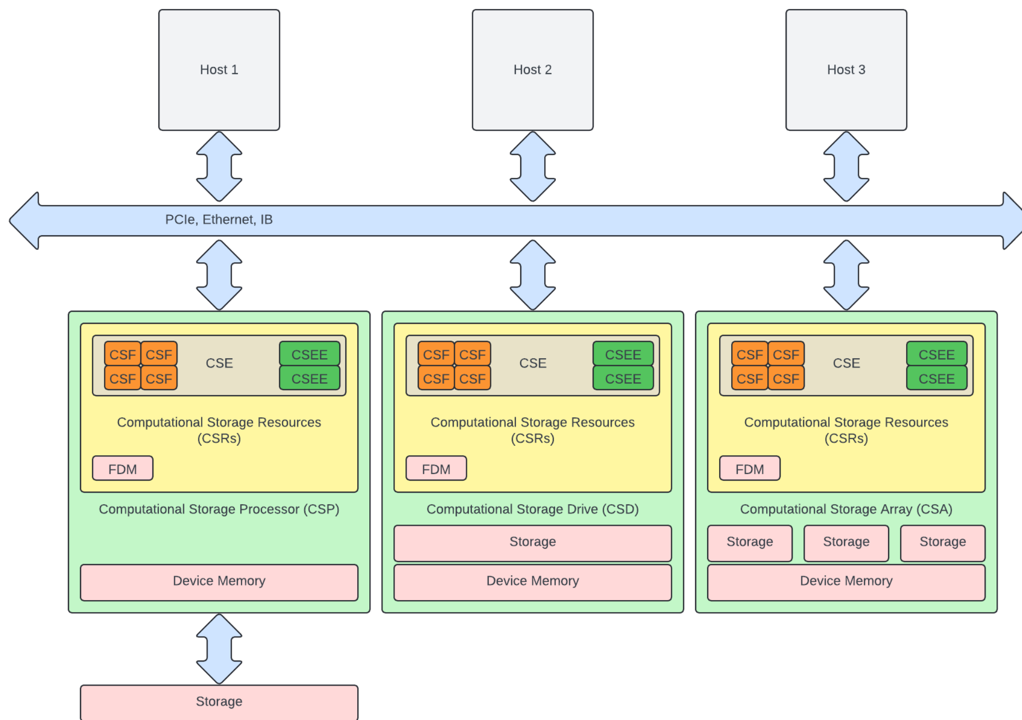


FIGURE 3.1: A diagram showing the relationship between the SNIA computational storage terminology.

- **Computational storage engine environment (CSEE)**: An operating environment for a computational storage engine. A computational storage engine environment could be a VM, accelerator, or any other unit with computational capabilities.
- **Computational storage function (CSF)**: Specific operations that may be configured and executed by a computational storage engine. A computational storage function is a concrete, device-specific or host-offloaded procedure compatible with at least one of the execution environments embedded within the computational storage device.
- **Computational storage processor (CSP)**: A device that provides computational storage functions for an associated storage system without providing persistent data storage. The main difference between computational storage processors and computational storage drives is the fact that computational storage processors do not have persistent storage. Instead, they are connected to persistent storage via PCIe, Ethernet, IB, or similar interfaces.
- **Computational storage resource (CSR)**: A resource available for a host to provision on a computational storage device, enabling that computational storage devices to be programmed to perform a computational storage function. Computational storage resource is a catch-all term for any subsystem on a device that supports computational storage capabilities.

- **Computational storage device (CSx)**: A computational storage drive, computational storage processor, or computational storage array.
- **Function data memory (FDM)**: Device memory used for storing data that is used by the computational storage functions and is composed of allocated and unallocated function data memory.

Furthermore, SNIA introduce a set of example mechanisms for program offload to computational storage:

- Bitstream based offloading for FPGAs, where hardware acceleration modules are synthesised and deployed to the FPGA. This approach deploys static functionality, which cannot be altered without reprogramming and rebooting the device.
- Operating System-based offloading, where the storage controller is deployed alongside an operating system. The operating system can take any form, from small and custom operating systems to fully-fledged ones like Linux. The operating system approach to offloading is practically static, as redeployment entails recompilation and redistribution of changes to code and configurations.
- Container-based offloading, where the computational storage controller is deployed via containers and can be redeployed or reconfigured on the go.
- eBPF based offloading, where the computational storage controller can execute eBPF bytecode. The programs can be shipped from the host or remotely via the network. This offloading mechanism is the most dynamic of the four, due to the ability to offload fully-fledged programs on the go. These programs can contain loops and even functionality with multiple distinct paths depending on runtime information.

In an interview with Dave Landsman of Western Digital, among others, he states that SNIA probably selected eBPF as the preferred offload mechanism because it is more straightforward to establish a standard ecosystem with architecture independence [56]. In other words, a host would not need to know the processor embedded in an NVMe device to offload, because it can use hardware-agnostic code. Furthermore, Jim Harris of Intel states that since eBPF already has a track record of efficiently handling network packets, it makes sense to expand the eBPF ecosystem to other use cases.

Theory of Operation

SNIA describes computational storage as the integration between compute resources and storage. With this architecture, it is possible to improve the performance of applications and infrastructure by freeing up capacity on existing processors, memory, storage, and I/O systems. Furthermore, these specify that compute resources can be placed directly within or between the host and storage.

Memory Management

As briefly mentioned in this section, SNIA introduces the concept of computational storage engines (CSR) and function data memory (FDM). A computational storage resource is where the computational storage function and the computational storage engine environment may be stored. Function data memory is the working memory of the computational storage function.

Function data memory is located on the computational storage device but may be mapped to a host memory address. Function data memory is dynamic and managed using the `csAllocMem()` function and the `csFreeMem()` function.

Since a computational storage resource may be anything from function data memory to vendor-specific resources, it is hard to clearly describe the memory management of the SNIA standard. This vagueness occurs because it is not clearly defined how to interact with non-function data memory memory.

Program Management

SNIA distinguishes between an Operating System environment, a Container Platform environment, a Container environment, and an eBPF environment. However, these are merely examples and not a proposed standardisation of environments.

Programs are either device-specific or, depending on the execution environment, offloaded from the host. The `csCSFDownload()` function allows offloading computational storage functions to appropriate computational storage engine environments.

Limitations

Since the specification of SNIA focuses on terminology and relationships rather than implementation details, we need answers to fundamental questions. For example, how do execution environments interact with storage? eBPF and containers are run in isolation by default, meaning that some mechanism to access storage must be introduced. Furthermore, we are left with questions on how to handle memory management efficiently. How do we avoid fragmentation in the function data memory regions? How do we guarantee cache coherency between host and device and execution environments like FPGAs and ASICs?

3.4.2 Non-Volatile Memory Express

In 2023, NVMe released the first iteration of the *Computational Programs Command Set Specification* [39]. This specification defines two new NVM namespaces/command sets.

Theory of Operation

The Computational Programs Command Set defines the mechanism for offloading programs from the host to an NVMe subsystem. These programs, which serve specific and well-defined purposes, can be defined by the device or downloaded from the host.

The specification allows programs to operate on data stored in one or more memory namespaces. Input and output data for these programs are managed through the memory namespaces using NVMe commands issued by the host. The specification introduces new functionality through I/O commands to execute programs and administrative commands to load programs, manage program activation, and manage memory range sets. Additionally, it defines data structures and log pages, including Identify Controller data structures and Identify Namespace data structures.

Memory Management

Memory Management, as seen in the Theory of Operation, is based on namespaces. More specifically, NVMe proposes having distinct buffers for programs and data. The programs are stored in one or more compute namespaces. These namespaces act as a bank, where the host issues transfers to the particular namespace and gets an identifier in return. This level of indirection supports offloading various programs with different sizes since the memory regions can be allocated dynamically.

Data slots function similarly, albeit with notable differences. They are located in a Subsystem Local Memory (SLM) Namespace that can source data from zero or more NVM Namespaces.

Since resources can be distributed over many namespaces, NVMe introduces *Reachability* as a mechanism to verify access to a resource. Reachability Associations is a tuple of two namespaces which can access each other's resources.

The specification introduces the notion of Memory Range Sets. A Memory Range Set consists of one or more ranges of SLM, each defined by an NSID, an offset, and a length. They are created within a single compute namespace, restricting program access to specified SLM areas.

Memory Range Sets are linked to programs via Execute Program commands and identified by a unique Memory Range Set ID within each compute namespace. The host manages these IDs using the Memory Range Set Management command, which is unique but not necessarily sequential. Different Memory Range Sets can overlap to allow multiple programs to share data, but ranges within a single set cannot overlap.

Compute namespaces can contain multiple Memory Range Sets independent of sets in other namespaces. Each compute namespace supports specific range granularity for Memory Range Sets. Memory Ranges are identified by IDs, with 0h referring to the data buffer, and others assigned sequentially within the set.

Program Management

Programs have several notable characteristics, including whether they are offloaded from the host or device-defined and their type, which can, for example, be eBPF or vendor-specific. Each program may also have a Program Unique Identifier (PUID). These characteristics are linked to a Program Index (PIND) within a specific compute namespace.

The I/O Command Set includes commands for loading, unloading, activating, deactivating, and executing downloadable programs and commands for activating, deactivating, and executing device-defined programs. Programs operate on data in SLM, are activated with the Program Activation Management command, and executed with the Execute Program command.

A compute namespace may support downloaded programs of specific types and may also have device-defined programs. Information about the supported program characteristics for a namespace can be obtained from the Identify data structures, the Downloadable Program Types List log page, and the Program List log page.

Program activation prepares a program for execution on a specific namespace by reserving the necessary compute resources. Depending on the program type, activation might involve tasks such as JIT compilation of an eBPF program or flashing an FPGA with an RTL program. Separating the preparation of the execution environment from the actual program execution helps ensure more predictable execution latency.

A namespace may host more programs than can be activated simultaneously, allowing for a higher number of device-defined and downloaded programs. The host can then choose which programs to activate. Before executing a program on a compute namespace, it must be activated on that namespace. The host uses the Program Activation Management command to activate or deactivate programs.

The I/O Command Set Identify Namespace data structure can be used to determine the limits on the number of programs that can be activated on a specific namespace. The Get Log Page command, specifying the Program List log page, can determine which programs have been activated.

3.5 Conclusion

Computational storage and program offloading have gained significant attention since the end of Dennard scaling in the mid-2000s. The main focus is to design and implement architectures to improve processing capabilities and throughput in high-performance systems by utilising hardware components other than the primary processor.

Kossmann and Franklin's article from 1998 on data shipping versus function shipping shows the benefits of a hybrid model that combines both methods [31]. This conclusion has been verified by Voruganti et al. in 2004 [58].

Meanwhile, eBPF offload is a contemporary form of function shipping, where queries and procedures are compiled into eBPF code and shipped to accelerators or computational storage devices.

In networking, program offloading benefits from standardised IPv4 and IPv6 protocols. This standardisation trivialises architecture and design, as the input and output formats are well-defined. This characteristic contrasts with storage devices, which are more complex due to the lack of standardised I/Os, and since they can vary significantly in their implementation and capabilities.

In this chapter, we have examined ten previous proposals for computational storage to understand the diverse design choices and their limitations. These include FPGA-based solutions like Willow, BlueDBM, INSIDER, REGISTOR, POLARDB, and NASCENT; ARM-based Biscuit, software-driven YourSQL, as well as specialised hardware architectures like OX and Eid-Hermes. Each proposed architecture has given us some understanding of the landscape of computational storage, with a tendency towards FPGA as the preferred hardware solution.

The Storage Networking Industry Association (SNIA) has proposed mechanisms such as bitstream-based offloading for FPGAs, OS-based offloading, container-based offloading, and eBPF-based offloading. Each of these mechanisms has different challenges and opportunities and varying degrees of reconfigurability. The NVMe *Computational Programs Command Set Specification*, introduced in 2023, describes how NVMe devices should handle program offload while introducing new terminology like computational programs, a form of discrete functional units, and the use of compute namespaces.

After surveying the design landscape of computational storage, we understand the complexities and challenges in performing and standardising program offloading to storage. We have learnt that these challenges arise due to the diverse and incompatible nature of drives and filesystems. Despite these complexities, we have seen promising standardisation attempts laying the groundwork for designing and implementing computational storage devices.

More specifically, we note the following challenges:

1. **Memory Management**: Memory management, including state management, differs significantly between systems. Some systems implement a functional approach, where state and data only live for the operator's lifetime. Other systems provide limited memory to handle their respective use cases. We generally see a tendency for memory to be restricted and limited, probably to avoid issues with dynamic allocation and cache coherency.

2. **Specialised interfaces**: The surveyed systems use vendor-specific interfaces and programming models. In essence, to offload computations or instructions to the systems, one must express them in the respective

structure of the device without the ability to reuse programs on other architectures.

3. **Restricted use-cases**: The surveyed systems target particular use-cases, including database systems, query execution, or file abstractions. The complex storage domain likely limits systems to target particular interfaces over generic, general-purpose interfaces.

4. **Hardware trade-offs**: While we observe tendencies towards using FPGAs as underlying hardware architecture, the survey makes it clear that there is no silver bullet in terms of hardware. Systems relying on FPGAs tend to be faster but more specialised and limited, while CPU-based systems support a more comprehensive range of use cases at the cost of performance.

Chapter 4

OpenSSD Platform

The *OpenSSD project* emerged in 2011 as an initiative to promote research and education on the recent SSD technologies [55]. The OpenSSD project has resulted in multiple platforms, including Jasmine, Cosmos, Cosmos+, Daisy and Daisy+. These boards enable researchers to develop and experiment with SSD firmware on actual boards. The boards are manufactured and sold by CRZ Technology with a license from Hanyang University.

The *Jasmine OpenSSD* was the first board developed within the OpenSSD project and presented in 2011. It features an *Indilinx Barefoot* controller with 96 kilobytes of SRAM, an *ARM7TDMI-S* core running at up to 87.5 MHz, a SATA 2.0 host-device interface, and 8 NAND flash memory slots.

The *Cosmos OpenSSD* replaced Jasmine in 2014 and is equipped with *HYU Tiger3* controller, which is implemented using a *XC7Z045-FFG900-3 Zynq-7000* FPGA. The controller has two 1 GHz ARM Cortex-A9 cores and 1 gigabyte of DDR3 SDRAM. Furthermore, the Cosmos has several PCIe connectors and SO-DIMM NAND flash slots.

The *Cosmos+ OpenSSD* replaced Cosmos in 2016 and came with a newer generation of the Tiger controller, the *HYU Tiger4*. The board is identical to the Cosmos platform, but the new controller offers NVMe capabilities.

The *Daisy OpenSSD* replaced Cosmos+ in 2019 and features a Xilinx Zynq Ultrascale+ MPSoC with four ARM Cortex-A53 cores running at 1.5 GHz, 2 gigabytes of LP DDR4, two 2x100GE and PCIe Gen3 x16 connectors, and a backplane interface for connecting to two M.2 SSDs [12, 23].

The *Daisy+ OpenSSD* replaced Daisy in 2021 and features a board similar to the Daisy, but with an NVMe Host Controller, NAND Flash Controller, and an FTL [14]. Furthermore, it has an ARM Mali-400 MP2 Graphics Processing Unit and 4 gigabytes LP DDR4 (DDR4-3200) memory.

This chapter discusses the opportunities and limitations of using the OpenSSD project as a testbed for computational storage. We will focus on the Daisy OpenSSD for the remainder of the chapter.

FIGURE 4.1: The Daisy OpenSSD [15].

4.1 Interface

The Daisy OpenSSD offers multiple methods of connecting to the host.

4.1.1 PCIe

The Peripheral Component Interconnect Express (PCIe) protocol operates on a transactional architecture, enabling communication through requests and responses [5]. It works with isolated physical connections arranged as lanes, with a packet-based data link protocol on top. Each lane has a pair of unidirectional, serial, point-to-point links.

Devices following the PCIe standard may have one or multiple memory-mapped spaces called Base Address Registers (BARs). These regions on the device are directly accessible from the host memory.

With PCIe, larger data transfers between the host and device can be facilitated independently of the host CPU through DMAs. These transfers occur at the hardware level via the DMA controller within the PCIe Root Complex. On the Daisy platform, DMA operations are handled by XDMA, which organises lanes into write channels (H2C) and read channels (C2H).

Figure 4.3 shows the lifecycle of a DMA request. Initially, a host process initiates a request to transfer a memory block to the device. Next, the DMA controller is initialised and given a pointer to a memory buffer, along with its size. The CPU commands the peripheral device to start the transfer. The DMA controller interfaces with the device by providing pointers and control signals via the DMA registers. In the context of XDMA, these registers are located in BAR2. During the transfer, the DMA controller incrementally updates its internal registers for each byte until the transfer concludes.

It is worth noting that DMA represents the most efficient method for transferring more significant amounts of data across PCIe.

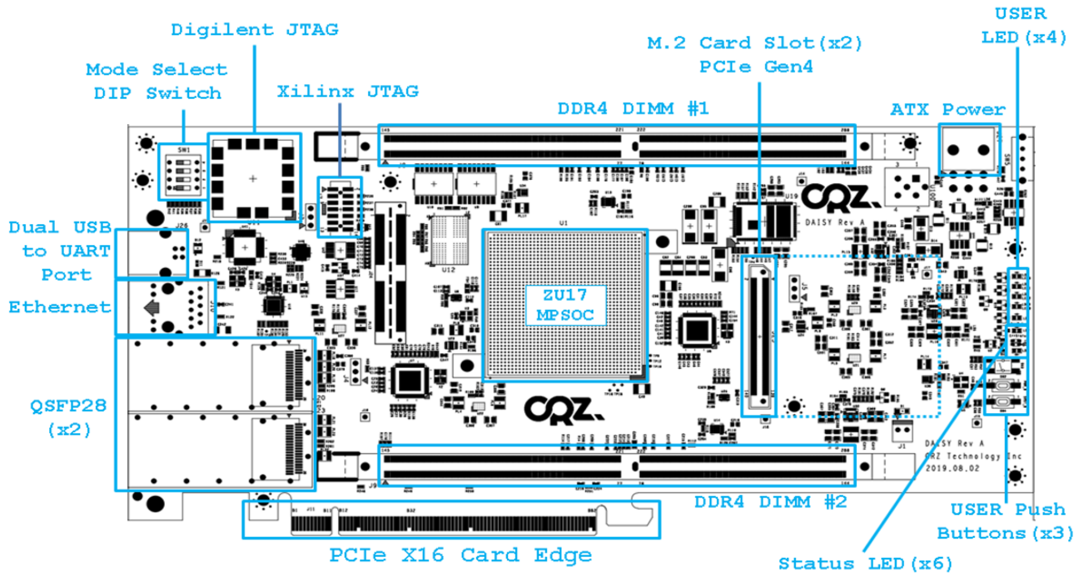


FIGURE 4.2: A diagram showing the physical components of the Daisy OpenSSD [15].

4.1.2 Ethernet

The Daisy OpenSSD has an RJ45 connector, offering 10/100/1000 ethernet connectivity to other machines. When running an operating system within the processing system of the MPSoC, this connection allows for an automatically configured connection, for example, via DHCP. Since this connection is limited to 1 gigabit per second, it is not suited for data transfers but rather as a control or debugging connection.

4.1.3 QSFP28

The Daisy OpenSSD comes with two QSFP28 connectors. Since QSFP28 has an upper bound of 100 gigabytes per second, these two connectors can be used to expose the storage controller over a network interface. This feature is especially relevant if the Daisy runs as a standalone device without a host.

4.2 Zynq UltraScale+ MPSoC

The Zynq UltraScale+ platform is a heterogeneous multiprocessing system, which integrates an ARM processor and an FPGA hardware accelerator [61]. The ARM processor, also known as the *Processing System* (PS), is capable of running entire operating systems, enabling the deployment of complex device controllers. The FPGA component, also known as *Programming Logic* (PL), enables the deployment of custom hardware components.

Zynq UltraScale+ MPSoCs come with multiple high-performance PS-to-PL ports, enabling connectivity between the operating system running on the PS and peripherals located within the PL. While the most efficient ports offer

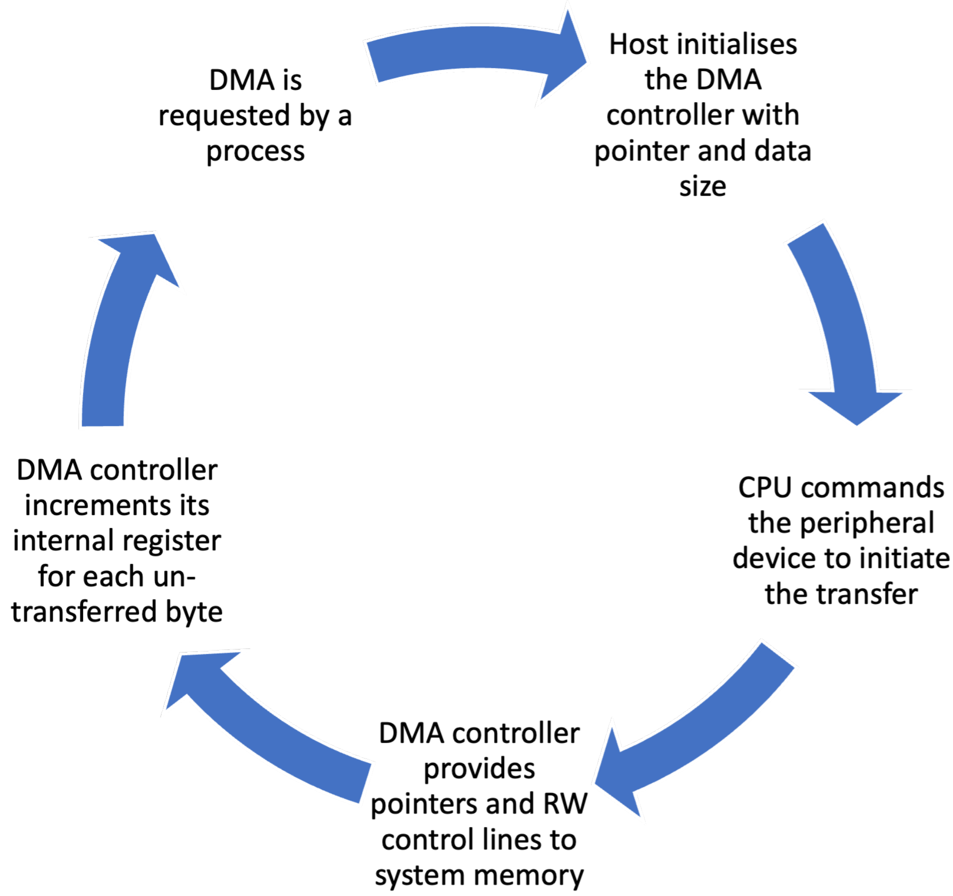


FIGURE 4.3: An overview of the steps of a single DMA transfer.

AXI interfaces, the chip also supports general-purpose ports such as GPIO and USB.

4.3 Software Development Kit (SDK)

Xilinx, acquired by Advanced Micro Devices (AMD) in 2023, was one of the major suppliers of programmable logic devices such as FPGAs. As mentioned in the previous section, the Daisy OpenSSD has a Zynq UltraScale+ MPSoC, which Xilinx developed. This dependency means that the SSD must be configured, built, and programmed using the Xilinx Software Development Kit (SDK), which we describe in this section.

The Software Development Kit consists of many IPs developed by Xilinx, the Vivado suite for designing the block design, Vivado HLS for developing IPs from higher-level languages like C, and the Petalinux SDK for configuring and deploying operating systems to the Xilinx boards.

It is worth noting that throughout this thesis, we will work with the 2019.1 version of the SDK, since this is the preferred version for deployment to the Daisy OpenSSD.

4.3.1 IPs

In FPGA programming, one primary task is to define and describe the hardware configuration. This description can take the form of Block Designs, and in the Xilinx SDK, it is stored as Hardware Design Files (HDFs) or Xilinx Support Archives (XSAs). With Vivado, the underlying Block Designs are expressed as a visual representation of how components within the device are interconnected, including any intermediary Intellectual Property (IP).

This section explores a subset of the typical IP modules often encountered in FPGA programming.

AXI

The Advanced eXtensible Interface (AXI) is the primary interface protocol for connecting Intellectual Property (IP) blocks [2]. AXI enables fundamentally different IP modules to connect via a general-purpose channel instead of relying on custom protocols or interfaces.

A practical example of AXI in use is a situation where a link is needed from the operating system residing in the Processing System (PS) with one or more FPGA peripherals. In this situation, we must utilise the PS-to-PL ports as described in Section 4.2. However, the configuration of the PS-to-PL ports on the Zynq often do not match the configurations of the AXI ports on the peripherals. To connect PS and the peripherals, or to directly link two or more peripherals, the easiest option is to add either an AXI Interconnect or an AXI SmartConnect to the block design. Both connectors have the same purpose of connecting AXI-compatible IPs together but with distinct differences. While the Interconnect allows for more fine-grained configuration, SmartConnects offer a more simplified and automatic approach.

PCIe DMA/Bridge – XDMA

The Xilinx Direct Memory Access (XDMA) IP enables the developer to configure the FPGA either as a Peripheral Component Interconnect Express (PCIe) Root Complex or Endpoint.

In bridge mode, XDMA operates as a PCIe Root Complex, connecting to underlying PCIe secondary devices [3]. In DMA mode, it instead exposes the device as a PCIe device with Direct Memory Access (DMA) capabilities to a host-machine [17].

On the Daisy OpenSSD, the physical pins connected to the host or underlying SSDs are wired to the PL part of the MPSoC. Connecting any XDMA IP with the board's processing unit in PS is optional. For example, in DMA-mode, the XDMA IP only connects one or more AXI backends to the PCIe link. The host can issue I/O operations to any of the apertures in these AXI backends with this configuration.

The XDMA IP has many configurable properties, which include the number of PCIe lanes, link speed, and Base Address Registers (BARs), to mention a few.

Lastly, XDMA enables the FPGA to trigger interrupts on the host via PCIe, supporting up to 32 legacy, Message Signaled Interrupts (MSI), and MSI-X interrupts. However, some of these interrupts are reserved for DMA operations. To trigger one of the interrupts, one must send a signal to a preassigned pin in XDMA IP. This signal must persist until the host acknowledges and services the interrupt, indicated on another pin on the IP.

It is worth noting that the ability for XDMA to operate independently of the on-board processing unit raises many open questions about memory and cache management. We discuss these challenges in Section 9.5.

MIG

The Memory Interface Generator (MIG) serves as a simple way to expose a Dual In-Line Memory Module (DIMM) slot as a memory aperture through AXI.

The reason that MIGs are necessary for accessing DIMM memory is simple: Modern DIMM sticks have 288 pins, each requiring precise wiring from the connecting processing unit or IP to the memory stick. Xilinx addressed this complexity by developing MIGs, simplifying the process by exposing these 288 pins as a single AXI connection. Without the MIG IP, developers would have to connect all 288 pins individually and develop a driver to communicate with the memory stick.

Configuring a MIG is done by inputting the attributes of the mounted memory sticks. This configuration includes, for example, I/O latencies, voltages, and detailed information regarding columns, banks, and ranks.

GPIO

The AXI General Purpose Input/Output (GPIO) IP module is a mechanism for managing individual pins in the Programmable Logic (PL) part of the MPSoC or external components beyond the FPGA. When connected to PS via the PS-to-PL ports, each pin becomes accessible through a pseudo-file within the `/sys/class/gpio` directory.

In Petalinux, every GPIO chip connected to the system is assigned a base, which is the identifier of the first pin on the GPIO chip. The operating system dynamically allocates this base during boot-time and can be deduced through various cues, such as the static memory address from PL.

Managing a pin from Petalinux is a simple process. First, one must export the pin, signalling the Linux driver to make the pin accessible to user-space. Afterwards, the pin's direction is set, typically to `out`, unless read-only.

Once the pin is configured, its state can be changed by writing either `0` or `1` to the value pseudo-file associated with the pin.

Below is an example showcasing the successful configuration of a GPIO pin, immediately toggled to `1`:

```
# Go to the GPIO SysFS folder
$ cd /sys/class/gpio

# Signal that pin 504 should be exported to user-space
$ echo 504 > export

# Signal that pin 504 is a read/write pin
$ echo out > gpio504/direction

# Toggle the pin by setting it to 1
$ echo 1 > gpio504/value
```

4.3.2 Petalinux

Petalinux is a part of the Xilinx Software Development Kit (SDK) developed for UltraScale+ MPSoCs to enable the configuration and compilation of a lightweight Linux-based operating system. The SDK takes a Hardware Description File (HDF) or Xilinx Support Archive (XSA) as input, and outputs configuration and system files, including device tree and drivers, to support the IPs in the block design. Petalinux allows users to enable a wide range of drivers and modules, including support for various drive types such as NVMe and Open-Channel SSDs, alongside software packages like GCC, GDB, and Python. Furthermore, developers can develop their own packages and modules, which are cross-compiled to integrate with the MPSoC environment.

To enable faster prototyping, the Petalinux SDK offers a set of "Hello World" programs and modules built for Petalinux with build configurations for languages like C and C++.

When compiling programs and modules, one challenge is the differences between the applications and libraries on the development machine, which may sometimes share a different architecture than the FPGA. To accommodate this architecture gap, Petalinux makes use of cross-compilation tools. Under the hood, Petalinux uses Yocto, an open-source collaborative initiative to develop tools and methodologies to generate and compile Linux distributions for embedded systems. The main idea of Yocto is to offer tools and processes that are architecture-agnostic, which guarantees compatibility across a spectrum of embedded hardware [59].

In practice, cross-platform compilation within Petalinux is hidden from the software developer. Programs and modules provide a Makefile without any architecture-specific configurations. This approach also enables developers to compile and validate code on the development machine without regard to

FPGA compatibility. When code is compiled through Yocto for deployment on the FPGA, Yocto appends a series of architectural configurations to the compiler. These include settings like `CROSS_TARGET` and the `-march` option, which abstracts away the intricacies of cross-compilation from the software developer.

To simplify working with Yocto, Petalinux offers a set of binaries for the configuration of projects. For example, the `petalinux-config` binary is used to access the underlying kernel and hardware configurations, thus enabling users to activate non-default drivers and modules. The `petalinux-create` command is a scaffolding command for creating templates for projects, custom modules, and applications.

To expand Petalinux with custom applications, modules or drivers, they are added to the `meta-user` part of the project specification and registered in the RootFS configuration file. After registration, the Yocto build mechanism will search for the BitBake recipes and compile the module or application following the recipe. For simpler kernel modules, the generated recipe and build configurations do not need to be changed and work out of the box.

Petalinux and Yocto allow several different recipe types.

1. App recipes: Recipes of the app type will be compiled and installed as a binary on the target operating system only with access to the use space APIs.
2. Board Support Package (BSPs) recipes: The BSP recipes are support recipes for adding code, drivers or configurations essential for operating the board. For example, the device-tree support package defines the target board's device-tree files (DTSs).
3. Core recipes: Core recipes provide information on what is needed to build a basic working Linux image. If Yocto already has the necessary information about dependencies and configurations, no core recipes need to be defined.
4. Kernel recipes: Kernel recipes integrate patches into Linux without directly altering the code. Due to software or hardware constraints, version locking Linux may also be necessary, thus not receiving bug fixes and security updates.
5. Module recipes: The module recipe is the kernel-space application version. These are configured and built to run within the kernel and can access all kernel-space functionality.

4.3.3 Vivado

Vivado is an application designed and developed to synthesise and analyse hardware description languages (HDLs). It also includes functionality to decrease developers' cognitive load.

One of the crucial features of Vivado is the ability to simplify the design phase of FPGA projects by automating the connection of IPs sharing similar interfaces. For example, when adding two unconnected AXI-compatible IPs, Vivado will suggest how to connect these two components and, if necessary, generate any intermediary AXI Interconnect IPs. As such, the primary task of an FPGA designer using Vivado is to understand in depth which IPs fits the use-case and define board-specific constraints.

In the context of Vivado, *constraints* refer to a set of rules connecting the block design, and thereby the FPGA, to external hardware and peripherals. Since this depends on the FPGA itself and the peripherals of the target board, automation is not feasible. However, Vivado may infer constraints where related constraints are explicitly defined, and the remaining wires or pins only have one valid solution. Since constraints depend on the target board, boards are often shipped with guides or specifications describing the port assignments for various hardware components.

When a valid block design and a valid set of constraints have been defined, Vivado will synthesise the RTL-specified design into a gate-level representation. After a successful synthesis phase, the resulting gate-level representation is implemented, that is, made into a board-specific bitstream. During this phase, Vivado can improve metrics like Total Negative Slack (TNS) and Total Hold Slack (THS). These two metrics indicate whether parts of the resulting implemented bitstream fail timing constraints, which could cause components to malfunction. For example, if the negative slack of clock paths is too high, the components driven by this clock pulse will start running at lower frequencies than synthesised. Vivado tries to improve the TNS and THS by moving components around to shorten paths and bring connected circuits closer together. It should be noted that THS and TNS issues often appear when block designs reach a size where the space of the FPGA is becoming limited, or when a block design is too complicated, for example, when many IPs are running on the same clock source.

4.3.4 Vivado HLS

Vivado is an application designed and developed for synthesising C programs into IPs using *High-Level Synthesis*. The main selling point of HLS is the ability to write simple high-level representations of functions and compile them directly to Verilog or VHDL.

With Vivado HLS, the developer can access an editor, where simple functions can be synthesised and packaged into a deployable IP. The editor also enables running a test bench on the IP before synthesising it, ensuring the C function has the intended functionality. While synthesising, the compiler generates Verilog or VHDL code matching the target MPSoC chip. As a parameter, it also takes the clock frequency that will be provided to the IP in the block design and the ability to define uncertainty in case the clock pulse is not fully reliable.

When writing functions in the Vivado HLS suite, it is possible to define the origin of parameters and meta-data using C `#pragma`s. For example, functions taking at least one parameter should define where the parameter comes from using `#pragma HLS INTERFACE`.

```
uint32_t filter(filter_t *in, filter_t *out, uint32_t num,
  uint8_t op, filter_t comp1, filter_t comp2) {
#pragma HLS INTERFACE m_axi port=in offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=num bundle=control
#pragma HLS INTERFACE s_axilite port=op bundle=control
#pragma HLS INTERFACE s_axilite port=comp1 bundle=control
#pragma HLS INTERFACE s_axilite port=comp2 bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

  ..

}
```

LISTING 4.1: An example of an HLS function with parameters and the correpsonding pragmas.

The function takes several arguments in Listing 4.1. For example, it takes two pointers, `filter_t *in` and `filter_t *out`, that point to a physical address of the `filter_t` type. In this function, the `filter_t` type is an alias of `uint32_t`. Using `#pragma`s, it is declared that these pointers are set using a slave interface, and that the data that the pointers point to should be accessed via the same AXI port, named `gmem`. When a pointer is declared to be set using the slave interface, the HLS compiler will create a set of registers accessible via an AXI Lite port on the IP called `control`. Next, the four parameters `num`, `op`, `comp1`, and `comp2` are declared to be set using the same AXI Lite slave port as the data pointers. Lastly, the `#pragma`s defines that the function's return value should be written to the same registers as the parameters.

After successful synthesis and export, the HLS IP can be made available to the other Vivado applications and inserted into a block design.

4.4 UDMA

UDMA or *u-dma-buf* allocates contiguous memory blocks within the kernel-space via the device tree [26]. We call these buffers for user-space DMA since UDMA makes them accessible from user-space.

First, we reserve one or more regions in the device tree. When UDMA has initialised and allocated a buffer, any user-space application can access it from user-space by opening the corresponding device file, e.g., `/dev/udmabuf0`, and mapping it to user memory space using memory mapping (mmap).

Furthermore, UDMA enables access to managing the CPU cache for the buffer's underlying memory regions. UDMA exposes functionality such as

cache flushing or invalidating all or parts of the DMA buffer while retaining cache enablement.

Without such cache management functionality, applications would either have to run in a cacheless mode, where all memory accesses are directed to memory, or manage the cache manually using the `dc` assembly instruction.

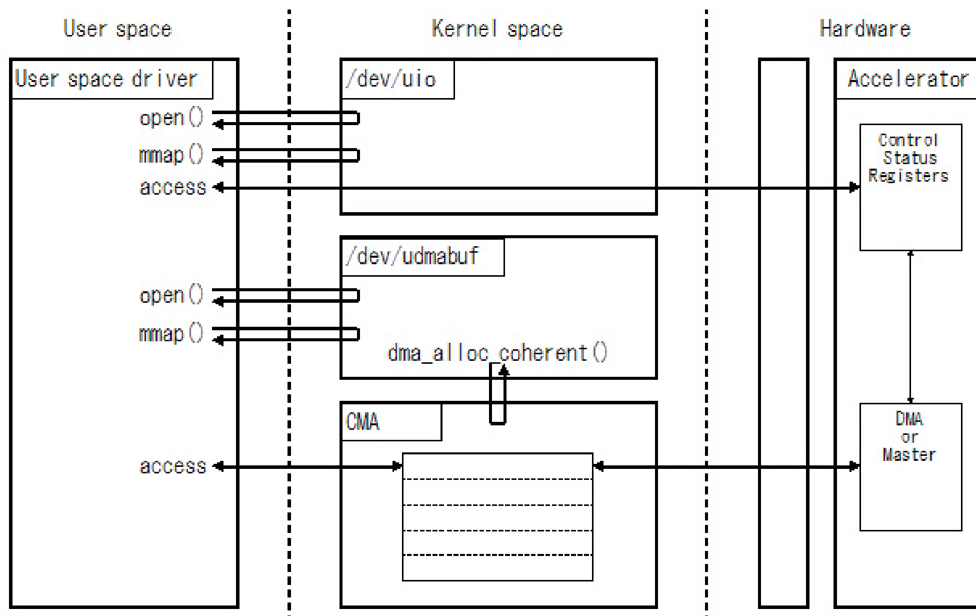


FIGURE 4.4: An overview of the architecture of the UDMA driver and how it interacts with the kernel [26].

4.5 Conclusion

The OpenSSD project has developed significantly since its introduction in 2011, with each generation of boards introducing new features for experimental storage research. As of 2024, the project has led to the development of five boards: Jasmine, Cosmos, Cosmos+, Daisy, and Daisy+. Jasmine, the first board, laid the foundation by supporting only basic SSD functionality and releasing the hardware design under an open-source license. Thereby, Jasmine provided a platform for innovation and research in SSD technology. The Cosmos board built upon this foundation by integrating more advanced controller technology and supporting a broader range of storage interfaces. Cosmos+ further developed these capabilities.

The introduction of the Daisy and Daisy+ significantly changed the OpenSSD project by transitioning from SATA interfaces to PCIe and NVMe. The Daisy board, which supports PCIe, Ethernet, and QSFP28 interfaces, enables low-latency connectivity and high-throughput data transfer. The board has a Zynq UltraScale+ MPSoC, a multi-processor chip with an ARM processor and an FPGA component. This combination allows for higher flexibility and more efficient storage operations.

The OpenSSD project has played and continues to play a critical role in experimental storage research. The OpenSSD project is the only project, to our knowledge, that offers an off-the-shelf hardware solution with high-speed, low-latency host connectivity, high-capacity DDR4 memory, NVMe M.2 connectors, and both an FPGA and general-purpose processing unit.

The alternative to the OpenSSD project is to rely on hardware that may have only some of the necessary hardware components and thus cannot be used for experimental research. Without high-speed, low-latency connectivity to the host, designing and experimenting with interfacing and drivers would be impossible. Without the high-capacity DDR4 memory, working with a significant amount of data and parallel tasks would be impossible. Without the NVMe connectors, non-volatile data on the device would be impossible. Without the FPGA and the general-purpose processing unit, program offload would not be possible.

However, it should be noted that using OpenSSD is not a free lunch. Adopting OpenSSD for experimental research means that the application or platform also inherits any limitations or challenges associated with the OpenSSD hardware or the internal Xilinx UltraScale+ MPSoC. Some inherited issues include cache incoherence, which can complicate data consistency between the host, storage device, and intermediate accelerators. Neither the OpenSSD project nor Xilinx mitigates this challenge, meaning it is up to the implementor to find a solution.

The fact that the UDMA framework is necessary to reliably expose memory regions to the processing system of the OpenSSD proves that memory management is a challenge. In essence, building any system on top of the OpenSSD will result in performance numbers and evaluations specific to the OpenSSD. This decision limits the generality of the experiments and the results. It only allows us to draw preliminary conclusions that should be tested or verified on other systems with different characteristics.

In conclusion, the OpenSSD project offers a suitable hardware platform for initial development and experimentation. However, the OpenSSD boards may not be suitable for production environments due to the inherent problems caused by using non-specialised hardware for specialised operations.

Chapter 5

Data Pipelines

With the recent emergence of artificial intelligence, machine learning, data science, and other data-intensive software systems, designing and developing efficient ways to handle large amounts of data have become necessary.

A data pipeline is a complex chain of interconnected activities, starting with a data source and ending with a data sink [38]. Data pipelines have become an essential tool for data-driven organisations and companies due to the ability to process many formats from distributed data sources with limited human intervention. A data pipeline streamlines the workflow by removing many manual activities. The automation and streamlining come from the ability to chain a series of sub-processes, where the output of one process is the input of another.

This chapter will describe the context of modern data pipelines and the frameworks and systems that are often used. Understanding the landscape of data pipelines is essential to understanding how data is used and integrated with underlying storage mediums.

First, we examine six popular systems for designing and implementing data pipelines. Afterwards, we examine a more recent approach, integrated data analysis pipelines, which unifies data management (DM) and query processing, high-performance computing (HPC), and machine learning (ML) training and scoring.

Ideally, the unified approach makes it easier to make complex data pipelines, as one extensible system can handle the whole pipeline rather than several connected ones. Furthermore, a unified system should yield opportunities for hardware acceleration and other optimisations.

5.1 DuckDB

DuckDB is an embeddable database system built in 2019 [50]. The main functionality is efficiently executing SQL queries within another process. Its primary focus is analytical workloads, unlike SQLite, which primarily focuses on transactional workloads.

DuckDB's design and implementation are based on its intended use cases. For example, DuckDB has a modular architecture with components such as a parser, a planner, an optimiser, and an execution engine. Furthermore, DuckDB can guarantee ACID through Multi-Version Concurrency Control (MVCC) implemented on read-optimised storage data structures.

DuckDB performs better than related systems like SQLite, MonetDBLite, and HyPer. The performance is dominant when processing large datasets on embedded hardware with constrained resources. Therefore, DuckDB's main selling point is handling large datasets efficiently in embedded or constrained environments.

5.2 TensorFlow

TensorFlow is a machine learning framework developed by Google Brain in 2016 [1]. It is designed to work with large amounts of data in heterogeneous environments. It is based on using dataflow graphs to represent data operations with a shared state. This graph can then be scheduled across potentially distributed nodes and computational devices like CPUs, GPUs, and TPUs.

Notably, Google uses TensorFlow in its infrastructure and has released it as an open-source project. These two circumstances have led to its widespread adoption in machine learning research.

Since the TensorFlow dataflow model simplifies computation and state management, it opens up new approaches for parallelisation. For example, it supports large-scale training and inference by utilising computational resources across multiple distributed systems.

The evaluation of TensorFlow's performance includes single-machine and distributed setup benchmarks. The results show competitive performance compared to related systems. The system's scalability is evaluated through experiments on image classification tasks, which showed performance improvements with increasing numbers of workers. Coordination mechanisms have been used to improve tail latency. Techniques like synchronous replication and backup workers have improved overall performance.

While widely successful, TensorFlow is still work in progress. The authors are still researching and focusing on automatic optimisation, memory management, fault tolerance, and dynamic computation structures to meet evolving user needs and challenges.

5.3 PyTorch

PyTorch is a machine learning framework introduced in 2019 [43]. It focuses on the middle ground between developer accessibility and performance. It offers a Python-like programming environment with support for hardware accelerators like GPUs. Notably, PyTorch has gained significant adoption in

academic research, with hundreds of submissions citing it at major conferences like ICLR.

The architecture of PyTorch is based on significant scientific computing trends. These include, for example, the development of domain-specific languages (DSLs), automatic differentiation, Python systems released under open-source licenses, and the accessibility of GPUs for parallel computing.

Lastly, PyTorch continuously aims to improve its speed and scalability. Plans include extending the PyTorch JIT to work efficiently outside the Python interpreter as well as improving support for distributed computation.

5.4 Spark

Spark is a framework introduced in 2012 to efficiently process data directly in memory on large clusters [60]. It is designed to overcome the limitations of current architectures like MapReduce. To do so, Spark keeps intermediate data in memory, drastically reducing the amount of data written to external storage. Furthermore, Spark's design is based on the Resilient Distributed Dataset (RDD), a fault-tolerant data structure that supports caching intermediate results directly in memory.

Spark is accessible through a simple API that supports typical high-level operations. It can integrate with multiple cluster programming models within a single framework, including MapReduce, SQL, and Pregel. Lastly, Spark supports interactive data analysis, enabling users to perform real-time queries directly from the Scala interpreter.

Spark's benchmarking efforts show that the system can outperform Hadoop by up to 20 times for iterative applications. It can also handle terabyte-scale datasets with low-latency queries. Spark is open-source licensed and has gained traction in the large-scale data analytics community.

5.5 Flink

Apache Flink is a data processing framework introduced in 2015 [8]. It is designed to focus on efficient stream and batch data processing. Its architecture is based on a distributed dataflow engine with programs represented as directed acyclic graphs (DAGs) of stateful operators connected by data streams. The engine has two main APIs: the DataStream API for processing unbounded data streams and the DataSet API for handling bounded data sets.

Flink supports multiple different notions of time. For example, it supports event, ingestion, and processing time, allowing for precise event correlation and out-of-order processing. The framework has a windowing system, producing early and approximate results while returning delayed and accurate

results later. This capability is attractive for real-time applications with eventual consistency, in particular.

Flink's ecosystem includes domain-specific libraries and APIs for machine learning (*FlinkML*), graph processing (*Gelly*), and SQL-like operations (*Table API*). These libraries generate programs that the core engine can later execute.

Flink is maintained by a broad community focusing heavily on academic research and practical production use cases. The combination of stream and batch processing capabilities and a broad community makes Flink a notable framework for data-driven applications.

5.6 Dask

Dask is a framework for optimising the scientific Python stack, introduced in 2015 [51]. Historically, the scientific Python stack has been limited to single-threaded execution with data residing in memory. Dask extends these capabilities by introducing efficient, memory-aware task scheduling, allowing existing tools like NumPy to handle complex and heterogeneous datasets while effectively utilising multi-core processors. Dask aims to provide parallelism and memory-aware scheduling without requiring researchers to rewrite their existing scientific Python code.

Dask uses "Dask graphs", which, like Flink, represent programs as directed acyclic graphs (DAGs) of tasks with data dependency information. These graphs are kept in memory during computation and use simple built-in Python data structures, such as `dictionaries`, `tuples`, and `callables`.

This approach allows Dask to implement parallel collections in Python, such as `dask.array`, which functions identically to the NumPy API but is more efficient for datasets that exceed memory limits. The execution of Dask graphs is managed by schedulers that determine the order of tasks based on runtime conditions. Dask supports multiple scheduling strategies, including single-threaded, multi-threaded, multi-process, and distributed execution. These strategies enable Dask to be efficient in small and large computing environments.

Dask also offers other parallel collection types, like `dask.bag` for efficiently handling collections of Python objects and `dask.dataframe` for large-scale data manipulation similar to Pandas. These collections and their respective schedulers make Dask a notable framework for data processing.

5.7 DAPHNE

DAPHNE proposes an architecture and implementation of an integrated data analysis pipeline. An integrated data analysis pipeline combines data management (DM) and query processing, high-performance computing (HPC), and machine learning (ML) training and scoring [16].

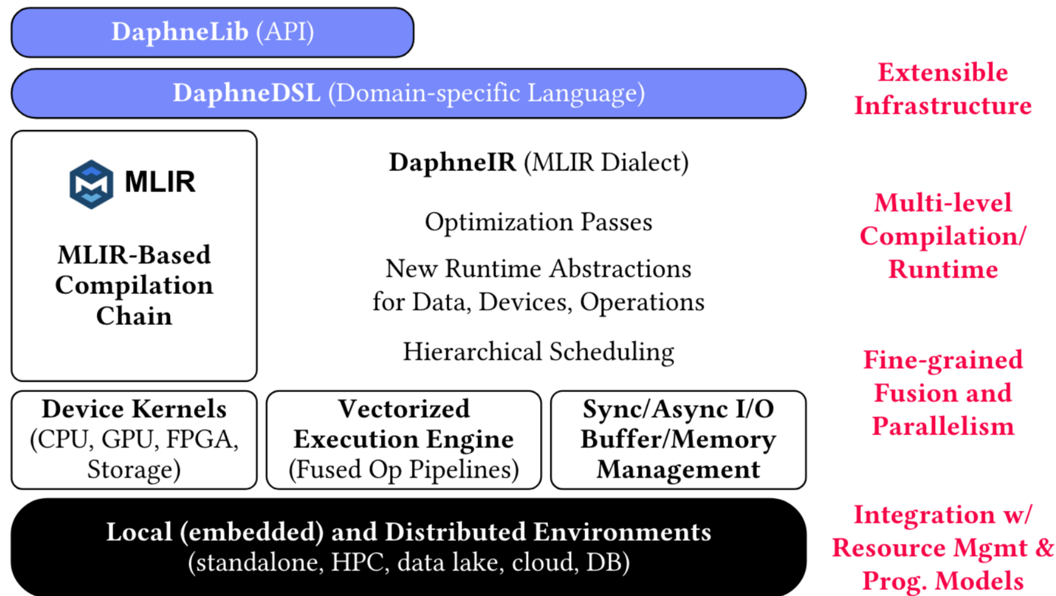


FIGURE 5.1: An overview of the architecture of DAPHNE, which is divided into four architectural tiers [16].

DAPHNE aims to streamline the development and execution of data pipelines by providing a simple, unified infrastructure instead of users having to navigate the diverse nature of existing tools and methodologies used in DM, HPC, and ML.

One of the critical challenges that DAPHNE tries to address is the integration of different programming paradigms, cluster resource management systems, data formats, and execution strategies. These components differ widely across DM, HPC, and ML domains. The system's architecture is based on the MLIR framework, which allows integration with existing applications and runtime libraries while still allowing extensibility for specialised data types and hardware-specific and hardware-accelerated optimisations.

DAPHNE's architecture includes principal components: language abstractions, a vectorised execution engine, multi-level scheduling, and extensibility for heterogeneous hardware devices and computational storage. The design aims to improve productivity and eliminate unnecessary overheads typically encountered in IDA pipelines. The vectorised execution engine, in particular, enables efficient use of computational resources by combining pipelines of frame and matrix operations and supporting local and distributed operations.

The initial experimentation of DAPHNE shows promising performance improvements over existing systems like MonetDB, Pandas, DuckDB, and TensorFlow. The experiments are based on multiple IDA pipelines, such as query processing with linear regression, earth observation data scoring, k-means clustering, and connected components analysis. The results indicate

that DAPHNE's integrated execution approach can benefit from hardware acceleration, computational storage, and vectorised operations.

One essential and notable element of the DAPHNE project and architecture is the focus on extensibility in the hardware acceleration layer. This design decision makes DAPHNE a prime candidate for exploring computational storage on a larger scale and a higher level.

5.8 Conclusion

A data pipeline connects processes from data sources to data sinks, streamlining workflows by automating tasks and reducing human intervention. We have looked at several essential frameworks used today to design and implement data pipelines.

We have studied seven frameworks for designing data pipelines, and they have given us a few interesting insights into the current state of the art. Firstly, the space is diverse, with several different characteristics and approaches. All systems have the same aim: to handle large amounts of data more efficiently. Some systems, like PyTorch, TensorFlow, and Dask, rely heavily on Python as the preferred language, regardless of the performance of Python[1]. Other frameworks like DuckDB try to improve large-scale data operations on heterogeneous and potentially resource-constrained hardware. Spark and Flink try to improve data throughput by limiting data movement and implementing efficient streaming. These systems and their respective characteristics indicate that researchers and data scientists prefer operating in convenient and known environments, even at the cost of significant performance decreases.

Lastly, DAPHNE aims to unify the diverse tools and methodologies across data management, high-performance computing, and machine learning. Combining the diversity of the six previous systems with DAPHNE reveals integration as an emerging optimisation approach.

We learned from DAPHNE that the trend towards integration opens up the simplification of resources, including memory. More specifically, with an integrated pipeline, the memory model is constant throughout the pipeline, enabling programs to save state and store data in long-lived caches. This resource management is not possible today since all systems utilise their memory model and data structures. For example, PyTorch and Dask rely on the Python environment, whereas Flink and Spark do not, which makes it non-trivial to share data management and data structures. The trend towards integrated pipelines also simplifies working with memory, as the developer only has to understand and interact with a single memory API.

[1] An experimental study has shown that Python uses 76 times more energy, executes for 72 times longer, and uses double the memory compared to C [45].

In conclusion, it is our understanding that integration with host-side systems must be made in a way that is easily accessible and has simple integration. Furthermore, we now understand that unifying data pipeline systems into one leads to attractive optimisation opportunities. DAPHNE, an emerging integrated data analysis pipeline, is a prime example of a contemporary framework that is accessible and extensible enough to expose complex hardware acceleration.

Chapter 6

Summary

Part I of this thesis discussed the state of the art in computational storage.

We discussed the Berkeley Packet Filter (BPF) from 1992, which enabled user-space programs to execute functions within the Linux kernel. The initial use case was network packet filtering. In 2014, Alexei Starovoitov introduced Extended BPF (eBPF) to modernise BPF for contemporary processors. However, it should be noted that eBPF lacks an official standardisation body, but the eBPF Foundation oversees its technical direction. eBPF bytecode is interpreted by the Linux kernel, with JIT compilers available for several architectures, including x86 and ARM.

uBPF offers a user-space library to the GPL-licensed kernel implementation of eBPF, making it more accessible for non-GPL projects. It features an RISC register machine with eleven 64-bit registers, a stack pointer, an implicit program counter, and a fixed-size stack. The uBPF VM can load and execute eBPF programs from a buffer of bytecode or an ELF file.

The end of Dennard scaling in the mid-2000s has significantly increased the research interest in computational storage and program offloading. The research focuses on designing architectures to increase processing capacity and throughput using hardware components other than the primary processor. Kossmann and Franklin's 1998 paper on data shipping versus function shipping shows the benefits of a hybrid model. This conclusion was verified by Voruganti et al. in 2004.

In networking, program offloading benefits from standardised IPv4 and IPv6 protocols, simplifying architecture and design due to well-defined input and output formats. In contrast, storage devices are more complex due to the lack of standardised I/Os and their varying implementations and capabilities.

We discussed several previous proposals of computational storage device implementation to understand the diverse design choices and their limitations. These include FPGA-based solutions like Willow, BlueDBM, INSIDER, REGISTOR, POLARDB, and NASCENT; ARM-based Biscuit, software-driven YourSQL, and specialised hardware architectures like OX and Eid-Hermes. These proposals tend towards FPGA as the preferred hardware solution.

The Storage Networking Industry Association (SNIA) has proposed four mechanisms for offloading, including bitstream-based offloading for FPGAs, OS-based offloading, container-based offloading, and eBPF-based offloading. Each mechanism has different sets of challenges and opportunities. The NVMe Computational Programs Command Set Specification, introduced in 2023, describes how NVMe devices should handle program offload.

Our survey of computational storage design clearly shows the complexities and challenges of standardising program offload to storage. Most notably, we see the challenges of working with diverse and incompatible drives and filesystems. Promising standardisation attempts may pave the way for future designs and implementations of computational storage devices.

The OpenSSD project has developed significantly since its introduction in 2011, with each generation of boards introducing new features for experimental storage research. As of 2024, the project has produced five boards: Jasmine, Cosmos, Cosmos+, Daisy, and Daisy+. Jasmine, the first board, introduced basic SSD functionality, and the hardware design was released under an open-source license. The Cosmos board continued the work by integrating more advanced controller technology and supporting a broader range of storage interfaces. Cosmos+ further developed these capabilities.

The Daisy and Daisy+ boards significantly improved the OpenSSD project by supporting PCIe and NVMe instead of SATA. The Daisy board supports PCIe, Ethernet, and QSFP28 interfaces, enabling low-latency connectivity and high-throughput data transfer. The Daisy has a Zynq UltraScale+ MPSoC, an ARM processor and an FPGA component. These boards allow for more efficient storage operations due to the higher flexibility.

Finally, we examined several contemporary frameworks used to design and implement data pipelines. Data pipelines aim to simplify workflows by automating tasks and reducing human intervention. DuckDB is an embeddable database system for efficient SQL query execution within another process. TensorFlow, developed by Google Brain, is a machine learning framework for large-scale operations across heterogeneous environments. PyTorch offers a balance of usability and performance, supporting hardware accelerators. Spark is a distributed framework for in-memory computations, designed to overcome the limitations of MapReduce. Flink provides high-performance stream and batch data processing, executing programs as a directed acyclic graph (DAG) of stateful operators. Dask enables parallel computation within the Scientific Python stack using dynamic, memory-aware task scheduling. DAPHNE presents an integrated data pipeline architecture that combines data management, high-performance computing, and machine learning, leveraging the MLIR framework for extensibility and hardware-specific optimisations.

We see a trend towards unifying data pipeline frameworks into integrated pipelines. Integrated pipelines are unified pipelines with a single memory

management method, a global scheduler for all operators, global data structures, and the ability to have long-lived caches or global state.

In summary, we make a set of critical observations in the state of the art.

1. **eBPF as a vendor-neutral instruction set architecture**: With the increasing popularity of eBPF and its several execution environments, it is positioned as a prime candidate for program offload to other domains, including computational storage. Our view is backed by both SNIA and NVMe, who propose eBPF as an instruction set architecture for storage. Today, computational storage architectures predominantly expose proprietary interfaces with restricted access to memory. With eBPF, exposing a vendor-neutral mechanism to offload is possible. Coupled with an open memory architecture, where memory is not limited in lifetime or significantly in capacity, computational storage drives can now execute more complex and long-term operators.
2. **Integrated pipelines are emerging**: The fact that DAPHNE proposes a unified and integrated approach to data pipelines gives a new perspective on computational storage. With an integrated data pipeline, the pipeline can manage memory on the host and computational storage from the source of the pipeline to the sink. The pipeline can cache data between operators or have a long-living state on the computational storage device in such architecture. This functionality is not possible, or at the least non-trivial, with today's systems due to diverse approaches to memory management.
3. **OpenSSD is now mature enough to support a complex computational storage device**: With the introduction of Daisy and Daisy+, the maturity of the OpenSSD project has reached a new milestone. These two boards are advanced and complex enough to house complex computational storage processors with enough resources to execute the operators described in bullet point 1. Theoretically, the Daisy boards can be equipped with 64 GB of DDR4 memory with a speed of up to 2400 MT/s or 19.2 GB/s. The boards can utilise either PCIe3 x16 at 16 GB/s or the two QSFP28 connectors with a total bandwidth of 25 GB/s. The four core processing capacities of 6 GHz provide a powerful executor for offloading operations to the device.

All three observations have one thing in common: a shift in the approach to memory management.

eBPF can operate on large memory buffers due to the use of offsets in memory operations, combined with the ability to be compiled from C. This approach enables programs to be written in a way where the location of the data is not known at compile time but instead represented as an offset of a pointer given at run time. The constant and long-living memory regions in integrated pipelines allow operators to have performance-improving side effects, including caching data for later use, at undefined points in time, or having device-specific states like read and write pointers. With OpenSSD

and the Daisy boards, we now have off-the-shelf hardware to support this novel approach to computational storage.

This proposed architecture, based on eBPF and the Daisy OpenSSD board, is called *Delilah*.

Part II

Contributions: Delilah

Chapter 7

Requirements

Several requirements shape the design and implementation when developing a novel computational storage device targeted at host-side integrated data analysis pipelines. This chapter outlines these requirements in detail.

1. **Memory Management**: As described in Section 3.1, traditional storage architectures typically ship data from the storage device to the host processor. In the same section, we described how advanced storage systems differ from the historical notion of function versus data shipping due to the asynchronous I/Os with side effects. As such, our novel computational storage device must not only implement shipping functions to a processing unit closer to the storage device itself, but also implement memory management functionality to support the complex I/Os. Furthermore, a well-designed memory management architecture enables us to integrate with host-side integrated data analysis pipelines. This requirement entails having mechanisms to share and synchronise memory regions between the host pipelines, the device processor, and I/O libraries on the device, as well as guaranteeing cache coherency between these. Memory management should be exposed to and manageable by the host, such that the integrated data pipeline on the host can store cached data and state it on the device without memory management becoming a bottleneck.

2. **eBPF Support**: With the emergence of eBPF (extended Berkeley Packet Filter) as the preferred Instruction Set Architecture (ISA) for function shipping across various domains, such as networking, storage, and kernel operations, our computational storage device must fully support this architecture. Essentially, this enables a host or remote application to directly ship eBPF code to our computational storage device for execution. This requirement enables the host-side data pipelines to structure operations in a vendor-neutral instruction set architecture, thus not binding the application to Delilah. The use of eBPF further answers how to structure operations between host and device. eBPF is also well suited for integrated data pipelines, as operators can either be pre-made and automatically compiled, or generated within the integrated data pipeline to match the workload precisely, increasing flexibility.

3. **eBPF Side-stepping**: After exploring the landscape of computational storage, including the standardisations efforts described in Section 3.4, we understand the difference between device-specific and offloadable programs. While eBPF offers significant advantages for code shipping, it also presents certain limitations and challenges. As a general-purpose ISA, certain functionalities and procedures may not be suitable for compilation to eBPF. For instance, integration with underlying storage systems and accelerators like TSL (Template SIMD Library) or HLS (High-Level Synthesis) may be non-trivial. We consider it a requirement that our novel computational storage device has a mechanism to invoke device-specific functionality directly from offloadable eBPF programs without interacting with the host. The host may offload some application-specific functionality while side-stepping to device-specific programs to execute procedures not expressible in eBPF, or too device-specific for the host to know at compile time. This requirement also simplifies the use of eBPF for the host-side data pipelines by creating a mechanism for abstracting away device-specific functionality that the host does not know. Without side-stepping, the pipelines must be able to express all instructions, including device-specific ones, before the device is known, which is infeasible.

4. **Contemporary Hardware Support**: Designing a hardware platform for computational storage typically requires a significant investment of resources. Our approach involves building a computational storage processor (CSP) and deploying it onto a readily available, contemporary hardware platform to avoid such investment. Such a platform should have the components to support function shipping, such as a Central Processing Unit (CPU), internal or external storage devices, and connectivity options for remote host interaction via PCIe or QSFP (Quad Small Form-factor Pluggable). Furthermore, these hardware platforms should be readily accessible off the shelf, enabling fast prototyping and deployment of our computational storage processor.

5. **Efficient Interfaces**: Since a critical purpose of computational storage is to increase the overall performance of a system, the interface for interacting with the computational storage device must be efficient. It should implement an asynchronous execution model, allowing the host or remote application to continue local processing while the execution runs. Furthermore, it should minimise unnecessary memory copies and, if possible, avoid the CPU for large data transfers. This requirement is essential, as data pipelines depend on efficient processing with asynchronous submission/completion interfaces. Any synchronous or inefficient interfaces would slow down the data pipeline, thus rendering computational storage impractical and purposeless.

To summarise, the novel computational storage device needs to support eBPF-based function shipping with support for complex memory management, with the ability to side-step to another execution environment

when complex functionality, beyond eBPF capabilities, is needed. It should be built on a readily accessible, off-the-shelf hardware platform and offer interaction through an efficient, asynchronous interface.

Chapter 8

Design

The five requirements specified in the previous chapter were learnt from surveying the computational storage landscape. The requirements guide the design of a novel computational storage processor. We will outline the design decisions in this chapter.

Lerner and Bonnet distinguish two types of PCIe architectures [34]. Firstly, On-Path enables the definition of new storage interfaces by placing the computational storage processor between the host and stored data. On the other hand, Off-Path architectures place the computational storage processor outside of the path to the stored data. Here, data movement is dictated by the host based on legacy interfaces. Contrary to NVMe, we approach the architecture as PCIe On-Path, where eBPF ultimately dictates any data movement.

We base the design decisions on a few assumptions about how integrated data analysis pipelines work. Firstly, we assume integrated data analysis pipelines can manage memory for the entire pipeline duration, from source to sink. Secondly, we assume that integrated data analysis pipelines are asynchronous. Lastly, we assume that the integrated data analysis pipelines are divisible into operators that can be offloaded onto computational storage.

Using the DAPHNE project as a point of departure, we see that all three claims hold. DAPHNE may allocate, deallocate, and manage memory at any point in the pipeline's flow. DAPHNE furthermore exposes task queues for execution on near-SSD CPUs or FPGAs. Lastly, DAPHNE is based on a domain-specific language, where any operator or function call has at least one specific implementation, which is extensible. These operators could be extended to have parts of the execution on the computational storage device while enqueued into the task queues.

8.1 Design Decision 1: Memory Management

We implement distinct memory regions for data and programs to ensure our computational storage device can run programs on its local processing

unit. This decision involves building the processing system and its surrounding infrastructure with compatibility in mind. The device's memory management architecture must support accesses from eBPF without any specific alignment or access pattern constraints.

Because the device has a local processing unit, separate from the host processing unit, any communication or command sent to the device involves some degree of state management between the host and the device. This information includes the current state of execution, return values, and error handling. To facilitate state sharing, we implement a mechanism, ideally a low-latency control register or memory region, for efficient communication with the host.

On the target hardware, the Daisy OpenSSD, we have two kinds of DDR4 memory: two DIMM slots in PL and low-power DDR4 in PS. Additionally, we may deploy BRAM in PL as necessary.

We rely on PL DDR4 memory for program and data slots accessed from the host via Direct Memory Accesses (DMAs). This decision is based on the significant capacity difference between PS and PL memory, which would cause capacity issues if slots were placed in PS. However, using PL memory, which is farther from the CPU, may increase latency.

We introduce a special kind of data slot, called the shared data slot, an isolated data slot unmappable by the host. This slot is placed alongside other slots in PL memory and is suitable for holding state, e.g., I/O pointers and cacheable files, from the underlying storage mediums to avoid read amplification.

Control registers are stored in the PS low-power memory, close to the CPU and accessed through one or more Base Address Registers (BARs). Since control registers do not require much capacity, we opt for the memory location closest to the processing unit to minimise latency.

To ensure that the device's local and host processing units can both map the memory regions, we make them static in size and location. This way, neither the host nor the device controller has to identify buffers after enumeration.

Static memory regions, both in capacity and location, simplify integration into host-side data analysis pipelines by eliminating the need for continuous buffer allocation and deallocation. This continuity allows the pipeline to operate, assuming data remains available in the buffers until explicitly overwritten.

The target hardware's varying memory characteristics, such as capacity and latency, are essential factors. When evaluating the novel computational storage processor, we will conduct experiments to compare the performance of different components.

8.2 Design Decision 2: uBPF

Given our requirement for executing eBPF within the controller of our computational storage device, we must find or develop an eBPF-based VM. Due to the stricter demands of programs targeted at the kernel and network systems, we cannot base our device on those. However, as outlined in Sections 2.2.1, 2.2.2, and 2.2.3, three significant execution environments exist outside the kernel and network devices.

Recall the three environments:

1. **hBPF**: A hardware implementation of an extended Berkley Packet Filter (eBPF) execution environment on FPGA. hBPF supports extending the functionality by utilising the `call imm` instruction. Furthermore, it has full test coverage for the supported hardware targets. The development targets are Arty and Zybo devices.
2. **uBPF**: An execution environment designed and built for eBPF execution in user-space. uBPF has support for both interpretation and JIT compilation of eBPF programs on x86-64 and ARM64 architectures.
3. **rBPF**: Similarly to uBPF, rBPF operates within user-space and has a cross-platform eBPF interpreter and a JIT compiler for x86-64. Originally derived from uBPF, it was reworked entirely in Rust.

Considering its ability to run directly within FPGAs, hBPF is considered the prime candidate as an execution environment for our computational storage device. However, there are certain limitations to acknowledge. hBPF is built and designed for Arty and Zybo devices, explicitly stating in its README that these devices are not meant to compete with multi-core accelerator cards, like the Xilinx Alveo [49]. Later design considerations will reveal that neither Arty nor Zybo devices are suited for our computational storage processor.

Given the choice between uBPF and rBPF, uBPF was preferred due to the authors' proficiency in the C programming language.

8.3 Design Decision 3: eBPF Sidestepping

Switching between offloadable and device-specific programs is essential for the flexibility and efficiency of the device. We propose using the `call imm` instruction of eBPF to signal to the VM or JIT compiler that execution should be handed over to a device-specific program. These programs, which we will call registered functions, are registered with the uBPF VM.

At the time of designing the computational storage device, it was noted that uBPF lacks out-of-the-box support for the `call imm` instruction. However, supporting this instruction merely involves accepting `R_BPF_64_32` as a valid relocation (in contrast to `R_BPF_64_ABS64`, which is the default supported relocation type). This challenge was deemed trivial, so the choice of uBPF as the execution environment remained unchanged.

8.4 Design Decision 4: Daisy OpenSSD

As outlined in requirement four, it is considered critical that our contribution is a computational storage processor (CSP) deployable on readily available hardware, in contrast to a fully-fledged computational storage drive (CSD).

While the Zybo Z7 device, supported by hBPF, is equipped with a Xilinx Zynq 7000, it cannot interface with high-performance storage devices or connect to a host or remote application via PCIe or QSFP.

In contrast to Zybo devices, the OpenSSD devices are ideal for experimental exploration, as they are purpose-built for storage research and experimentation. At the time of device design, the latest iteration of the OpenSSD project was the Daisy OpenSSD.

The Daisy OpenSSD has a processing unit, two NVMe slots, and QSFP and PCIe connectors, which perfectly align with our requirements.

Given our choice of building a computational storage processor on top of the Daisy OpenSSD, opting for uBPF over hBPF is a natural choice.

8.5 Design Decision 5: Extend Eid-Hermes

Since the emergence of eBPF beyond kernel and network architectures is recent, limited publicly available projects exist for offloading and executing eBPF code on accelerators, storage or other embedded systems.

Eid-Hermes is an exciting public proposal for offloading eBPF to accelerators. However, the project has yet to be successfully deployed on non-emulated and non-virtual architectures.

Unfortunately, as of 2024, the Eid-Hermes project can be deemed End-of-Life (EOL) due to the absence of updates and bug fixes since 2021. Despite the author of this thesis having submitted several bug reports alongside proposed fixes, they have yet to be merged as of June 2024.

Eid-Hermes presents a promising protocol for eBPF offloading to a computational storage device via PCIe. However, due to the abandonment and lack of development, Eid-Hermes has several critical areas for improvement. In order to align with requirement five, we will now outline the necessary changes to the Eid-Hermes protocol.

8.5.1 io_uring

Eid-Hermes assumes that data transfer and program execution should be run in a synchronous environment. However, since we deploy our computational storage processor (CSP) to the Daisy OpenSSD, it can do multiple parallel transfers and executions. As such, we must modify the Eid-Hermes protocol to avoid using synchronous system calls such as `open`, `close`, `ioctl`,

`write`, and `read`. Instead, our version of the Eid-Hermes interface should use io_uring, an asynchronous I/O API for Linux created by Jens Axboe.

Another critical motivation for abandoning synchronous system calls is to avoid context-switching to the kernel. Context-switches are costly since they temporarily store the application's state, execute some operations in kernel-space, and restore the application state. Instead, with asynchronous APIs like io_uring, the execution is kept in user-space, thus removing the context-switch costs and further increasing performance.

When applications schedule transfers and executions with io_uring, they must prepare a *Submission Queue Entry* (SQE) with information about the request. The applications should use the opcode of the SQE to `IORING_OP_URING_CMD` to indicate that they are submitting a command to the underlying computational storage device. Then, due to the nature of io_uring, they can specify the operation in further detail by setting the `cmd_op` field.

After SQE submission to the computational storage device via io_uring, the host application can perform other work and either continuously poll for a *Completion Queue Entry* (CQE) or wait for completion in a blocking manner. Due to the architecture of io_uring, the CQEs do not contain information about what operation has finished. However, the host application can use the data field of the SQE to identify it. The data field of the SQE will be passed back via the CQE.

In summary, whereas Eid-Hermes uses a synchronous interface based on system calls and requires context-switching to the kernel, our device will expose a thread-safe, parallel, and asynchronous interface to execute multiple operations concurrently. This asynchronous interface also fits nicely with the typical integrated data pipeline, where operations are scheduled without expecting them to be completed immediately. Instead, completions are handled as they come, allowing the host to work on other tasks while waiting.

8.5.2 Shared Data Slot

Our device should also differ from the Eid-Hermes interfaces in terms of what the executed programs can access. In Eid-Hermes, a program is given access only to a *data slot* that only has a guaranteed lifetime spanning the duration of execution and can only be accessed by one execution at a time.

However, as described earlier in the memory management design decision, we introduce a shared data slot. Without the shared data slot, it is impossible to perform stateful parallel operations, and the system's performance is significantly reduced.

8.6 Summary

The design decisions for the novel computational storage device are well-aligned with the specified requirements and designed for integration with host-side integrated data analysis pipelines.

The design has distinct memory regions for data and programs to support complex I/O operations and asynchronous data processing. Using PL DDR4 memory for program and data slots supports high-capacity memory regions with the ability to do Direct Memory Accesses (DMAs) from the host. This design choice makes it easy for host-side pipelines to transfer data back and forth asynchronously. We place control registers in low-power PS memory to minimise latency, thus decreasing the time for host-side applications to trigger executions. The memory architecture is designed to support efficient integration with host-side integrated data analysis pipelines.

The choice of uBPF as the execution environment for eBPF programs ensures compatibility by avoiding tight coupling between the controller and the execution environment. This decision aligns with the requirement to support eBPF, enabling host-side data pipelines to structure operations in a vendor-neutral instruction set architecture without knowing the device capabilities in advance.

Implementing the ability to switch between offloadable eBPF programs and device-specific functions circumvents the limitations of eBPF. Using the `call imm` instruction to trigger device-specific programs guarantees that complex and device-specific tasks can be handled without being expressible in eBPF. This approach simplifies the integration with the host-side data pipelines, as these pipelines do not need an in-depth knowledge of the underlying device characteristics. In essence, this design choice simplifies and abstracts away device-specific functionality.

By selecting the Daisy OpenSSD as the hardware platform, we gain access to a device with a processing unit, NVMe slots, and QSFP and PCIe connectors. This decision avoids the significant resource investment required for developing custom hardware while ensuring the device can be readily deployed and prototyped using off-the-shelf components.

The extension of Eid-Hermes to use io_uring for asynchronous I/O operations enables us to use an off-the-shelf protocol while significantly improving throughput and latency. The design maximises performance by avoiding synchronous system calls and context-switching, which is necessary when integrating into data pipelines. A shared data slot allows the device to store stateful data between executions, thus decreasing read-amplification and enabling partitioning of operators. These interface improvements ensure that the computational storage device can handle multiple concurrent operations efficiently.

Chapter 9

Implementation

In this chapter, we describe the design and implementation of Delilah, an architecture for efficient eBPF-based offload to computational storage, based on the previous chapters' requirements and design. We call this architecture Delilah, and it was first introduced in the D6.2 deliverable of the Horizon 2020 DAPHNE project in 2022 [6]. It was subsequently presented at the workshop Data Management on New Hardware (DaMoN) in Seattle, USA, in 2023 [23].

Figure 9.1 gives an overview of how the components of the Delilah architecture are connected. The architecture has three distinct areas. First is the host-side area, which consists of an application interacting with the Delilah driver. The Delilah driver interacts with the second area, the PL region of the device, which is supported by an FPGA and contains hardware components. This area is also where hardware-accelerated functions live. The PL area is connected to the PS area, where the Delilah controller resides, and where the eBPF code is executed.

9.1 Board Configuration

Deploying Delilah to the Processing System of the Xilinx MPSoC requires us to use Petalinux to configure and deploy an operating system.

9.1.1 Delilah Controller

Delilah's controller logic and block design are encapsulated in a Petalinux distribution. We have based this distribution on the Petalinux configurations of the `Daisy_M.2_PCIe_MIG_201901_20210413` block design by CRZ Technology [11]. The changes to the original configurations are focused on memory management and quality-of-life changes to developers of Delilah.

First, Delilah is added as a user-space application recipe in the project-specification part of Petalinux. This recipe provides two underlying Yocto BitBake definitions, one in Release mode and one in Debug mode. The only difference between the Release and Debug modes is the automatic stripping of debug symbols. In essence, the compiled binary in Debug mode

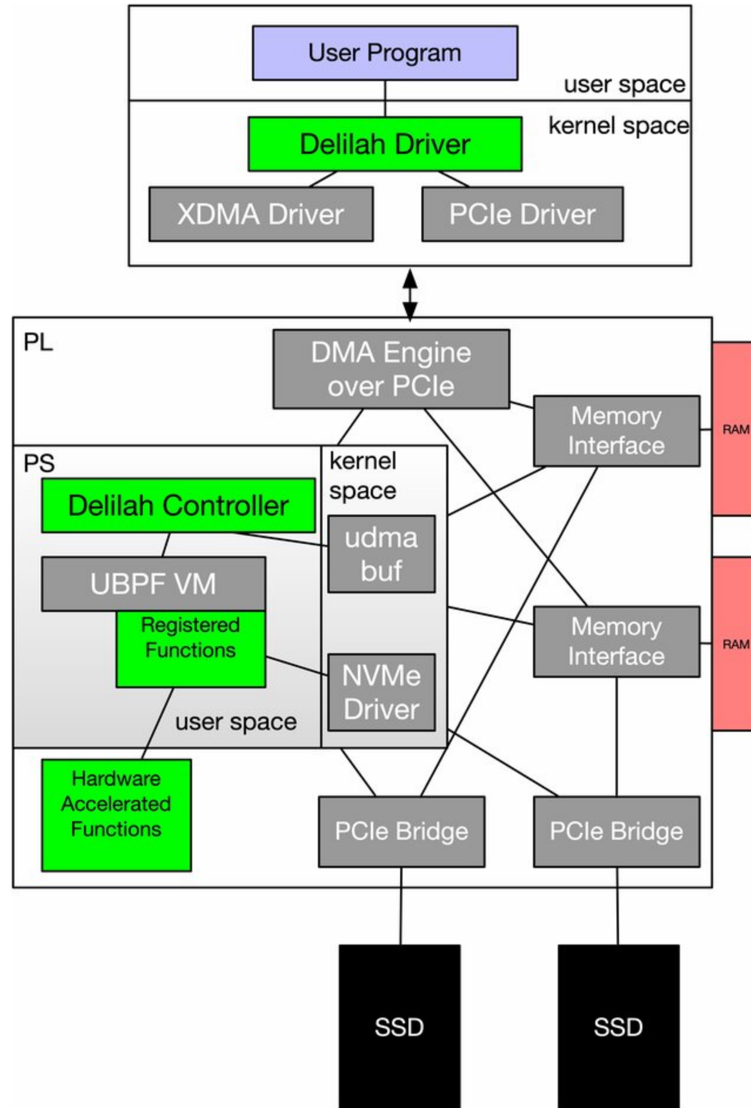


FIGURE 9.1: Overview of the proposed Delilah architecture [23]. The blocks coloured light green denote components belonging to Delilah, and blocks in grey denote components from other vendors.

will interact better with tools like GDB at the cost of less optimisation and higher binary size.

Both BitBake recipes declare all files required by Delilah, both headers and implementations, uBPF, TSL, and any static files needed for Delilah to run. The only static file needed in Delilah is the `.profile` file, which helps Delilah launch on the board automatically. Since the board starts multiple terminals for different interfaces, the script merely checks that the current invocation is on the JTAG debug session, namely `ttyPS0`. This check means that developers accessing the Daisy OpenSSD via minicom or similar tools will see Delilah's output and can restart or interact with the controller.

```
#!/bin/bash

```

```
3 # Get the terminal device name
4 TERMINAL=$(tty)
5
6 # Check if the terminal is /dev/ttyPS0
7 if [ "$TERMINAL" = "/dev/ttyPS0" ]; then
8   /usr/bin/delilah
9 fi
```

LISTING 9.1: The contents of .profile

9.1.2 Device-tree

Besides the Delilah controller, Petalinux also compiles and configures various other aspects of the Daisy OpenSSD needed for deploying Delilah. For example, the device tree has been modified explicitly to define memory regions for allocation by UDMA and to expose the hardware-accelerated filtering engines via user-space IO (UIO).

```
1 delilah_bar: delilah@0 {
2   compatible = "shared-dma-pool";
3   reusable;
4   reg = <0x0 0x10000000 0x0 0x2000000>;
5   label = "delilah_bar";
6 };
```

LISTING 9.2: Declaration of the region reserved for the BAR0 register.

```
1 udma_bar {
2   compatible = "ikwzm,u-dma-buf";
3   device-name = "delilah_bar0";
4   size = <0x0 0x2000000>;
5   memory-region = <&delilah_bar>;
6   dma-coherent;
7   sync-mode = <3>;
8 };
```

LISTING 9.3: Declaration of the consumer of the region reserved for the BAR0 register.

Listing 9.2 and Listing 9.3 show the declaration of reserved memory region in the device tree. In the first listing, it is declared that the 32 MB from `0x10000000` is reserved and labelled as `delilah_bar`. It is marked that this region is reusable, thus mappable by the Linux kernel, and should be added to the shared DMA pool. This approach is the preferred way of reserving memory regions on Xilinx boards. The second listing takes the previously declared memory region and assigns it to a consumer. In this example, it is declared that the UDMA driver will consume it, and that the region is DMA-coherent. It is marked as DMA-coherent since the physical location of the region is in PS memory, where the hardware automatically manages cache coherency using cache-coherent interconnects. Setting sync-mode to 3 declares that if the `O_SYNC` flag is *not* provided on access to the region, the region must fall back to CPU caching.

9.1.3 ECC initialisation

The two `M393A2K40BB1-CRC` DDR4 DRAM memory modules mounted in the Daisy OpenSSD are Error Correction Codes (ECC) memory. ECC is a mechanism to protect against undetected data corruption and is most often used in applications and systems where such data corruption is intolerable. It is, for example, expected to use ECC in database, financial and research applications.

Electrical and magnetic interferences can cause bits inside DRAM memory modules to flip to the opposite state. A protection against such flips is parity bits, where one bit per word is used as a checksum. In the case of a bit-flip, it is possible to reverse and repair the word back to the original value.

Since DRAM memory is volatile, the contents of the DRAM module at boot-time are undefined. As such, any ECC parity check will fail and cause the operating system of the Daisy OpenSSD to go into a fatal failure state. To prevent the kernel from failing, we have, in collaboration with the OpenSSD manufacturer, implemented an initialisation routine that correctly initialises ECC. This mechanism is patched into the `board/xilinx/zynqmp/zynqmp.c` file of u-boot, and will iterate through and write to the entirety of the ECC-enabled memory on boot to ensure all words have a correct parity bit.

9.2 Protocol

As previously stated, we design our device around Eid-Hermes. However, Eid-Hermes has never been successfully deployed to an actual accelerator. While experimenting, the author discovered several protocol defects that must be alleviated to guarantee stable operations. We will outline those changes in this section.

Due to these protocol changes, our device will become incompatible with the original Eid-Hermes driver.

9.2.1 64-bit support

The original Eid-Hermes driver assumes all memory buffers reside in a 32-bit addressing space. This assumption must be revised, as many accelerator boards expose a physical addressing scheme. When relying on a 32-bit protocol, we can only access the first 4 GB of the physical addressing scheme. On some Xilinx FPGAs, the location of more significant memory regions is limited to specific memory ranges, often outside of the first 4 GB of addresses. Our device must change the protocol, allowing memory region locations to be 64-bit if necessary.

`ehcmddone`	`ehcmdexec`	State
0	0	engine never started.
1	0	engine finished running.
0	1	engine should start and have never run.
1	1	engine should start and have previously run.

TABLE 9.1: The distinct states of Eid-Hermes/Delilah engines.

9.2.2 ehcmddone race-condition

On the original Eid-Hermes driver and protocol, two bits indicate the state of an execution engine. The `ehcmddone` bit indicates whether a given execution engine has finished executing, and `ehcmdexec` indicates whether the execution engine should start.

The Eid-Hermes protocol, where `ehcmddone` is solely writable by the device, has an unintended side effect. Specifically, in cases where an engine has previously finished, the Eid-Hermes driver interprets the `ehcmddone` bit as indicating that execution has already finished without considering the `ehcmdexec` bit.

To address this race condition, we have two potential solutions: either make the `ehcmddone` bit dependent on the `ehcmdexec` bit, or reset the `ehcmddone` bit when a new command is submitted to the engine. We opt for the latter approach in our design, as it offers a more semantically accurate representation of the device's internal state. By requiring the device to clear the bit before execution, there is an observable period during which the device is in a semantically invalid state, signifying that it is both done and executed concurrently.

9.2.3 Interrupt masking

Eid-Hermes is designed with an incorrect assumption regarding the configuration of the Xilinx XDMA IP on the device, deviating from the Xilinx specification. By default, the XDMA IP disregards any interrupts transmitted from the device to the host via PCIe. However, if the host-side driver can receive and handle interrupts from the device, it must set the mask bits of the XDMA register space accordingly. It must set the bits to one for the interrupt IDs to accept and zero for those to ignore. For drivers utilising libxdma, this process can be automated using `xdma_user_isr_enable` and `xdma_user_isr_register`. In our design, these functions should be invoked in both the setup and teardown paths to enable delivery of interrupts when execution has finished.

9.3 Driver

As described in Chapter 8, parts of Delilah are based on Eid-Hermes. The host-side driver is built on the same structure as Eid-Hermes but with some

notable differences. Both the drivers of Delilah and Eid-Hermes are Linux character device drivers.

Since Eid-Hermes is based on the traditional filesystem system calls, the Eid-Hermes driver implements the `read_iter` and `write_iter`, which enables applications to utilise the `read()` / `write()` functions to write to the Eid-Hermes device. However, this poses some significant challenges, as `read_iter` and `write_iter` are synchronous and do not take any parameters that can be used to indicate the destination buffer. The Eid-Hermes driver also implements the `unlocked_ioctl` function to handle incoming commands.

Instead, the Delilah driver is based on io_uring commands and exposes only the `uring_cmd` function. Depending on what opcode the application sends to the driver, the driver either initiates a DMA request or forwards the command to the device. The driver uses two layers of queues: the outer io_uring queues, which hold all submitted commands, and the inner work queues, which abstract away synchronous calls in an asynchronous interface. When submitting device commands (e.g., execution of eBPF), the io_uring queue entry is submitted, the control register is updated immediately, and control is given back to the application. The queue entry is marked as completed when the corresponding execution engine raises an interrupt, which we will describe in detail later. Since this operation is asynchronous, there is no inner queueing mechanism. However, when a DMA transfer is enqueued, it will be either put into the Host-to-Card (H2C) or Card-to-Host (C2H) queue. The need for these queues stems from the fact that the XDMA transfers are synchronous, meaning that with the queue, we can return control to the submitting application while the inner queue worker handles the synchronous call. When the DMA is complete, we return a completion event from the worker. We can parallelise these operations as DMAs are executed on one of the four parallel H2C or C2H channels.

All queues are implemented using Concurrency Managed Workqueue (*cmwq*) [24]. The choice of using cmwq comes from the fact that cmwq handles multi-threading over CPU cores while supporting scalability in the form of automatic resource management. In this way, we can use high-priority kernel-controlled queues without worrying about the strict resource constraints of the kernel or how to schedule workers. Furthermore, cmwq also enables us to instruct the kernel always to have at least one worker ready to handle submitted DMAs.

The driver supports the following eight commands.

1. `DELILAH_OP_PROG_EXEC` signals Delilah to execute an already offloaded program. The program will be executed in interpreted mode, which we will discuss later in this thesis.

2. `DELILAH_OP_PROG_WRITE` signals Delilah to load a program to the device.

3. `DELILAH_OP_DATA_READ` signals Delilah to initiate a Direct Memory Access transfer from the device.

4. `DELILAH_OP_DATA_WRITE` signals Delilah to initiate a *Direct Memory Access* transfer to the device.

5. `DELILAH_OP_PROG_EXEC_JIT` signals Delilah to execute an already offloaded program. The program will be executed in Just-in-Time (JIT) mode, which we will discuss later in this thesis. The rationale for a distinct command for both interpreted execution and JIT execution is to avoid creating unwanted coupling between these two execution modes. As Delilah or uBPF develops, significant differences in parameters and characteristics between these two commands may arise.

6. `DELILAH_OP_CLEAR_CACHE` signals Delilah to clear any purgeable device caches. This operation makes experiments more reproducible and independent of previous operations on the device.

7. `DELILAH_OP_INFO` signals Delilah to emit device-specific information like controller version, number of execution engines, and buffer characteristics like physical memory addresses and sizes.

8. `DELILAH_OP_CLEAR_STATE` signals Delilah to clear the internal state. As we will see later in this thesis, Delilah exposes a memory buffer to hold the state between executions and execution engines. Since this state is not exposed or accessible by the host, the host can signal it to be cleared.

The `cmd` field of the SQE is populated with a pointer to a struct holding operation metadata. If the opcode is set to `DELILAH_OP_PROG_WRITE`, `DELILAH_OP_DATA_READ`, or `DELILAH_OP_DATA_WRITE`, the `cmd` field must point to a `delilah_dma` struct, which holds information about the local host-side buffer and the device-side buffer. For `DELILAH_OP_PROG_EXEC` and `DELILAH_OP_PROG_EXEC_JIT`, the host must populate a `delilah_exec` struct holding information on which engine to execute in, which program to execute, and which device-side working memory region to attach to the engine. Furthermore, the struct contains instructions as to if and how SCI[1] should operate. If the opcode is set to `DELILAH_OP_CLEAR_CACHE`, the `cmd` field must point to a `delilah_clear_cache` struct that describes which execution engine will be used to clear the cache. If the opcode is set to `DELILAH_OP_CLEAR_STATE`, the accompanying struct holds information about the execution engine and what part of the internal state buffer will be cleared.

The only exception to the rule that the `cmd` field must be populated to a pointer holding metadata for a command is the `DELILAH_OP_INFO` operation. The `cmd` pointer must point to a `delilah_device` struct that the driver and device will populate.

Both the Eid-Hermes and Delilah rely on the Xilinx XDMA IP. Both drivers are based mainly on Xilinx's own XDMA driver. The device is exposed as

[1]Selective Cache Invalidation is described in Section 9.5.

a file in the Xilinx driver, and the write offset corresponds to the device's physical address.

Eid-Hermes and Delilah differ by exposing a control register on top of this functionality, located in BAR0. This control register manages the physical addresses, such that the provided offset corresponds to an offset within a predetermined memory buffer.

We configure XDMA to use 8 PCIe lanes, divided into 4 H2C and 4 C2H lanes.

9.4 Device Controller

The Device Controller of Delilah is located in Petalinux, which, in turn, resides in the Xilinx MPSoCs Processing System domain. It consists of multiple modules separated into modularised header files and at least one implementation. This architectural structure enables future versions of Delilah to be built for other devices and backends. We describe all modules in detail in this section.

The device controller has only one non-modular file, the controller's entry point. This file holds the `main()` function, which is the first function called and which sets up the controller. First, it allocates a `delilah_t` struct, which holds all global state and metadata. Control is then given to the memory module to allocate memory, to the Hermes module to prepare the control register for device enumeration, to the IRQ module to spawn interrupt handlers, to the loader module to spawn execution workers, and to the hardware module to prepare any hardware-accelerated functions. As such, the controller has a simple architecture, where modules are in charge of their specialised area and called on demand.

9.4.1 Modules

Command

The command module of Delilah is in charge of defining and implementing the handlers of all supported commands. Command implementations follow the same signature, seen below.

```
1 typedef uint64_t handler_t(struct delilah_thread_t* thread,
2                           struct hermes_cmd_req* req,
3                           struct hermes_cmd_res* res,
4                           struct delilah_t* delilah);
```

LISTING 9.4: The signature of Command handlers in Delilah.

The first parameter of a handler contains information about the executing thread. For example, the handler can access the engine ID and the thread ID, as well as pointers of the control registers associated with the engine located in BAR0. Lastly, the thread struct provides access to the global Delilah state struct.

Although the command registers and the Delilah global state are accessible through the thread struct, they are also provided as parameters 2–4 for convenience.

```
uint64_t
delilah_command_clear_state(struct delilah_thread_t* thread,
                            struct hermes_cmd_req* req,
                            struct hermes_cmd_res* res,
                            struct delilah_t* delilah)
{
  uint64_t size =
    req->clear_state.size == 0 ? DELILAH_SHARED_SIZE : req->
   clear_state.size;
  log_debug("Clearing state %lld bytes, %lld offset", size,
            req->clear_state.offset);
  memset(delilah->shared + req->clear_state.offset, 0, size);

  res->status = HERMES_STATUS_SUCCESS;

  return 0x0;
}
```

LISTING 9.5: The simplest implementation of a command handler in Delilah. The Clear State command is in charge of setting the internal shared data slot to 0s.

Observe how the handler updates the `res->status` entry in the control register to `HERMES_STATUS_SUCCESS` to indicate to the host that the command has been completed successfully. It should also be noted how the union in the `hermes_cmd_req` struct is assumed to contain the entries of `clear_state`. The host driver is responsible for guaranteeing that the command request struct is formatted correctly and contains all necessary information.

Config

The configuration module of Delilah contains only macro definitions of the settings that are not controlled internally in Delilah. These macros include, for example, physical locations of slots and the number of engines to spawn, which should correspond to the number of logical cores and *semantic versioning* values to propagate to the host on initialisation.

To summarise, the configuration module is a convenient way to store all adjustable settings in one place.

Functions

The functions module is responsible for mapping the eBPF `call` instruction (see Table 2.1) from a single integer to executable code. The only product of the module is a `static struct` array of the `ext_func` type. This type contains an internal function index, a function name mapped to the immediate value in the binary, and a void pointer to a function.

When a C program is compiled to an eBPF binary, all function calls are converted into an integer and a function table is created in the resulting ELF binary. When a binary is loaded into Delilah, this function table is read by uBPF. On uBPF engine initialisation, the Loader module will read the `static struct` array from the functions module and allow uBPF to handle the `call` correctly.

Hermes

The Hermes module is deprecated and enables Delilah to adhere to the Eid-Hermes protocol. As discussed in Chapter 8, Delilah is no longer compatible with Eid-Hermes, nor the original driver. As such, the Hermes module name is a relic from previous development versions of Delilah.

The Hermes module contains several header files, all used to structure commands, requests, and responses in a structured way. For example, the `delilah_bar0` struct defines how the control register of BAR0 is structured. Furthermore, the `hermes_cmd_req` and `hermes_cmd_res` struct defines how requests and responses are organised in the protocol and how parameters for commands are passed to Delilah.

The Hermes module has a single implementation file, which implements the `delilah_hermes_configure(struct delilah_t* delilah)` function. This function is responsible for initialising the control registers with values from the Config module.

Hardware

The Hardware module maps certain registered functions from the Functions module to drivers of hardware-accelerated functions.

For example, the Functions module exposes a set of filtering registered functions (`delilah_hw_filter_*`). These functions are merely a facade for the hardware module. Most importantly, the five registered functions are mapped into a single hardware-accelerated function with different arguments.

The function within the Hardware module responsible for filtering, `delilah_hw_filter`, transforms arguments to match the input format of the HLS IP. For example, pointers provided by the eBPF program are always relative to the execution context, i.e., the logical addressing space of the data slot or the shared data slot. It is necessary to map and transform these addresses from logical to physical addresses. The Hardware module hands this responsibility to the Memory module, the only module with a full view of the memory configurations. At the same time, the Hardware module is also in charge of notifying the Memory module of any data hazards, intending to force CPU cache flushes or invalidation.

Another characteristic of the Hardware module is the coupling with Xilinx-generated drivers. The Vivado HLS platform automatically generates

C drivers that write to the correct offsets in the HLS registers. The Hardware module will automatically invoke the Xilinx driver to write any appropriate registers to set parameters, start execution, and fetch return values.

Interrupt

The Interrupt (IRQ) module exists to expose a mechanism to notify the host on command completion.

```
return_t delilah_irq_configure(struct delilah_t* delilah);
return_t delilah_irq_close(struct delilah_t* delilah);
return_t delilah_irq_raise(uint8_t id);
```

LISTING 9.6: The exposed functions of the IRQ module to be implemented.

Currently, the Interrupt module is only implemented by `gpio.c`, which opens a memory map to `0xB1000000`. This physical address hosts a GPIO module with four pins, which can signal XDMA to send an interrupt request to the host. We achieve this by spawning four threads, all held back by a `pthread_mutex_t`. When the `delilah_irq_raise(uint8_t id)` function is invoked, the corresponding thread is released and will raise the matching interrupt bit via the GPIO IP for approximately 10 microseconds.

Practically, the GPIO pins are raised via a memory-mapped region. The GPIO IP managing the pins is mapped to the address space located at `0xB1000000`. We change the nth bit at that memory location to raise an nth pin. For example, to raise the third pin and trigger the third interrupt on the host, Delilah must write `0b0100` or `4` to `0xB1000000`. Since all interrupts are triggered using this memory location, mutex locks are employed to ensure the writes are atomic.

It should be noted that the GPIO implementation of the Interrupt module deviates from Xilinx documentation. The implementation should keep the bit raised until the host acknowledges it. However, this acknowledgement happens too quickly for the GPIO IP to observe. The 10-microsecond waiting period is enough for the XDMA IP to correctly notice and handle the interrupt.

This module is one of the prime examples of why the strict modularisation in Delilah exists. The interrupt mechanism would undoubtedly be different if Delilah is ever deployed to other architectures or systems. For example, if Delilah were exposed to an ethernet interface, the interrupt module could be in charge of sending a specifically formatted completion network packet.

Loader

```
return_t delilah_loader_configure(struct delilah_t* delilah);
return_t delilah_loader_unload(struct delilah_t* delilah);
return_t delilah_loader_start(struct delilah_t* delilah);
```

LISTING 9.7: The exposed functions of the Loader module to be implemented.

The Loader module is the essence of Delilah. The Loader module exposes three functions that must be implemented. The first function, configure, is invoked during the initialisation phase of Delilah and is responsible for setting up an execution environment. In the default implementation of the Loader, four uBPF VMs are created, and the registered functions from the Functions module are registered. The registration of each function is where the term *registered function* originates.

The uBPF VMs in the Loader module are nearly identical to the IOVisor implementation, with two minor differences. Firstly, we have extended the relocation check in the `ubpf_load_elf` to not reject relocation type 10, synonymous with `R_BPF_64_32`. This minor change enables eBPF programs to call registered functions, thus allowing eBPF to sidestep and live up to design decision 3.

Secondly, we extend the `ubpf_exec` function to use the third and fourth registers. At execution start, these two registers will hold a pointer to the shared data slot as well as the size of the slot. This change enables stateful parallel executions, allowing operators to share information while running.

When the `delilah_loader_start` function is invoked, Delilah will spawn four threads, each with CPU affinity on their CPU core. Before controlling the thread, a struct of the `delilah_thread_t` is prepared. This thread identifies the individual threads and is passed to any executed commands registered in the Command module.

```
1  void *worker(void *p) {
2    struct delilah_thread_t *thread = p;
3    short num_cmds = sizeof(commands) / sizeof(struct command_t);
4    short opcode_ok = 0;
5
6    while (!thread->delilah->exiting) {
7      if (thread->ctrl->ehcmdexec != HERMES_CMD_START) {
8        usleep(1);
9        continue;
10     }
11
12     thread->ctrl->ehcmddone = HERMES_CMD_NOT_FINISHED;
13
14     log_debug("Command received for engine %i (op: %p).",
15               thread->engine, thread->cmd->req.opcode);
16
17     for (int i = 0; i < num_cmds; i++) {
18       if (commands[i].opcode == thread->cmd->req.opcode) {
19         commands[i].handler(thread, &thread->cmd->req,
20                             &thread->cmd->res, thread->delilah);
21         opcode_ok = 1;
22         break;
23       }
24     }
25
26     // opcode unsupported
27     if (!opcode_ok)
28       thread->cmd->res.status = HERMES_STATUS_INVALID_OPCODE;
```

```
29
30      thread->ctrl->ehcmdexec = HERMES_CMD_STOP;
31      thread->ctrl->ehcmddone = HERMES_CMD_FINISHED;
32      thread->cmd->res.cid = thread->cmd->req.cid;
33      opcode_ok = 0;
34
35      delilah_irq_raise(thread->engine);
36    }
37
38    pthread_exit(NULL);
39  }
```

LISTING 9.8: The implementation of the worker thread.

The code of the execution worker can be observed in Listing 9.8. When the worker starts, it will pull the registered commands from the Command modules and immediately go into an unbounded loop. This loop will only exit when the `exiting`-flag in the Delilah global state is set to true. The loop checks whether the command start bit is set to 1. If not, the worker sleeps for a microsecond and returns to the start of the loop.

If the start bit is set, the worker will set the done bit to zero and determine the appropriate command handler. The worker will loop over all registered commands to find a handler that matches the opcode of the request. If no handler is found, the response status is set to `HERMES_STATUS_INVALID_OPCODE`. If a handler is found, it will be invoked, and control is given to the handler. The handler, not the Loader, updates the response struct with return values and codes.

When the handler has finished executing, the Loader updates the control register to indicate to the host that the engine is no longer running, and that execution has finished. Lastly, the Interrupt module is called to notify the host that the command has finished being executed. It should be noted that the interrupt raise function returns immediately, while the interrupt handling is running in the background. This design decision enables the Loader to perform any necessary cleaning work without being blocked by the Interrupt module.

Memory

The Memory module connects Delilah to the memory regions used for data and program slot placement. As described in the design chapter, Delilah's memory architecture must support efficient access by the device, without particular access patterns or alignments, in a host-mappable way. The Memory module is essential since it handles buffer management and exposes guaranteed physical placement and contiguousness. This guarantee is a non-trivial challenge, as seen in Section 4.4.

```
1 return_t delilah_mem_alloc_bar(struct delilah_t* delilah);
2 return_t delilah_mem_alloc_data(struct delilah_t* delilah);
3 return_t delilah_mem_alloc_shared(
4     struct delilah_t* delilah);
```

```
5
6 return_t delilah_mem_sync_get(uint8_t type, uint8_t id,
7                               uint32_t size,
8                               uint32_t offset);
9 return_t delilah_mem_sync_set(uint8_t type, uint8_t id,
10                              uint32_t size,
11                              uint32_t offset);
12
13 return_t delilah_mem_unalloc_bar();
14 return_t delilah_mem_unalloc_data();
15 return_t delilah_mem_unalloc_shared(
16     struct delilah_t* delilah);
17
18 return_t delilah_mem_copy(uint8_t src, uint8_t dst,
19                           uint32_t size,
20                           uint32_t src_offset,
21                           uint32_t dst_offset);
22
23 uint64_t delilah_mem_virt_to_phys(uint64_t virt);
24 uint64_t delilah_mem_virt_to_slot(uint64_t virt);
25 uint64_t delilah_mem_virt_to_offz(uint64_t virt);
```

LISTING 9.9: The exposed functions of the Memory module to be implemented.

The Memory module exposes twelve different functions to be implemented. The first three are responsible for setting up and mapping the memory regions. In the UDMA implementation, this is done by opening up the respective character devices in `/dev` and memory mapping them into Delilah. The three *unalloc* functions do the inverse of the allocation function by unmapping the regions from Delilah and closing the file descriptors.

If Delilah is not launched with the `--static-shared` argument, it will not open the pre-allocated buffer on the PL memory intended for the shared data slot. Instead, Delilah will use `malloc` to allocate the shared memory region in PS memory. This functionality exists to experiment with and evaluate the resource consumption of placing memory regions in PS and PL memory.

The `delilah_mem_sync_get` and `delilah_mem_sync_set` functions are responsible for invoking the UDMA cache coherency functionality. This function is implemented by signalling to a particular file in SysFS. It should be noted that the memory synchronisation functions do not have to be implemented for cache-coherent memory regions.

The `delilah_mem_copy` function is a convenience function exposed to the eBPF programs, enabling them to copy parts of a data slot to another without knowing the underlying addressing scheme. It takes two integers representing the source and destination slot, a size to move, and two offsets representing the location within the two data slots. While data sharing is more efficient through the shared data slot, the memory copy function is an alternative.

The functions `delilah_mem_virt_to_phys`, `delilah_mem_virt_to_slot` and `delilah_mem_virt_to_offz` are all built to enable conversion from

logical memory-mapped addresses to physical addresses, slots, or offsets. For example, when calling the registered functions that initiate offloading to hardware accelerators, the buffer addresses are all logical addresses of the memory-mapped character device of UDMA. The functions will take the logical addresses and convert them to physical addresses. We can do this because the memory module knows both the logical and physical address of the buffer.

It is a systematic and straightforward process to convert a logical address to a physical one. To convert from logical to physical, we initially iterate over all data slots to see if the logical address falls within the range of any data slot. If it does, we will proceed with the physical address calculation. Firstly, we take the `HERMES_DATA_LOC` macro, which holds the location of the first program slot. We then add the cumulative size of program slots, giving us the physical address of the first data slot. Subsequently, we add the cumulative size of data slots before the matched slot to this address, giving us the physical address of the start of the matched slot. The offset is then calculated by subtracting the logical address provided to the function from the logical address of the matched data slot. Adding this calculated offset to the already known physical address of the buffer yields the resulting physical address. Determining the slot or offset alone is a subset of the process to determine the physical address.

Utilities

The Utilities module exposes convenience functions and macros. For example, the `errors.h` file defines all error codes of Delilah. The `time.h` file exposes functionality to easily measure the time it takes for blocks of code to execute. The `units.h` exposes macros to easily define memory sizes, e.g., by writing `4 * MiB` which expands to `4 * 1024 * 1024`. Lastly, the `log.h` and `log.c` expose a logging library to print information to the console easily. The library exposes six different log-levels, `trace`, `debug`, `info`, `warn`, `error` and `fatal`.

For now, the `trace` level is left unused. The `debug` level indicates state changes, e.g., execution of programs and management of cache coherency. The `info` level gives runtime information about Delilah's internal program flow. For now, it only indicates when Delilah has successfully started, and when Delilah has received a signal to terminate child threads and exit. The `warn` level indicates errors that stem from the host application and are not a problem caused by Delilah. An example could be offloading invalid eBPF programs or triggering hardware acceleration on an undefined execution engine. The `error` and `fatal` indicate internal Delilah errors. The `error` indicates that Delilah can continue operating, albeit in an undefined way, and `fatal` indicates that the error is significant enough to warrant immediate termination. Often, fatal errors stem from memory management issues, where Delilah cannot access the underlying memory buffers used for data and program slots.

Type	Seq. Read	Seq. Write	Random Read	Random Write
PS Memory	3.16 ns	3.12 ns	120.02 ns	434.75 ns
PL Memory	3.53 ns	2.26 ns	574.79 ns	574.70 ns

TABLE 9.2: The memory latency of accessing PS and PL, respectively.

TSL

The TSL, or *Template SIMD Library* module, is an experimental module for integrating Delilah with TSL. TSL is a C++ header-only library focusing on SIMD operations and offering hardware-agnostic use. The provided functionality directly maps to hardware or uses scalar workarounds for compatibility. Rather than a static library, it employs a Python-based generator, enabling customisation for specific hardware configurations. This approach ensures traceability and reduces code size, simplifying the management of redundant code inherent in a template library.

While experimentation with combining the strengths of TSL and Delilah is subject to future work, significant performance improvements have been observed when using ARM Neon to filter workloads in Delilah. This improvement will be discussed further in Section 12.5.

9.5 Selective Cache Invalidation

Using all available caching mechanisms and data access optimisation is crucial due to the nature of data processing, where data reads and writes are common. Previous work at the University of Illinois, Boston University, and the University of Waterloo showed a significant memory access latency to DRAM on Xilinx ZCU102 [4], the same chip as found on the Daisy board. The article shows that memory latency is around 3 nanoseconds for L1 access and 20 nanoseconds for L2 access. Furthermore, they observed that sequential memory accesses, even for larger data sets, were comparable to L2 accesses. However, random accesses are in the order of hundreds of nanoseconds. While hundreds of nanoseconds may not sound like much, Dean A. Klein of Microsoft presented numbers at WinHEC in 2007 that showed a typical normalised latency of memory accesses in the order of tens of nanoseconds, not hundreds [30]. As such, the ZCU102 has suboptimal memory access latency.

To counter this trend, we experimented with enabling CPU caching on Delilah. Enabling CPU caching speeds up data processing for sequential access. Table 9.2 shows the latency of accessing both PS and PL memory from Delilah with CPU caching enabled. Memory latency is around three nanoseconds for PS and PL memory when accessing memory in a sequential pattern. However, when accessing the memory in a random pattern, thus missing the L1 and L2 cache, the latency increases to over a hundred nanoseconds, as described in the Bansal paper [4].

File Size	Write Prog	Write Meta	Exec Prog	Read Result
1.00 KB	0.019 ms	0.006 ms	235.539 ms	0.066 ms
10.00 KB	0.032 ms	0.030 ms	235.541 ms	0.092 ms
100.00 KB	0.028 ms	0.020 ms	235.656 ms	0.199 ms
1.00 MB	0.033 ms	0.032 ms	237.082 ms	0.897 ms
10.00 MB	0.049 ms	0.019 ms	244.652 ms	11.295 ms
100.00 MB	0.021 ms	0.010 ms	320.455 ms	80.432 ms

TABLE 9.3: The runtime of offloading a program to Delilah with SCI disabled that reads a file from the underlying SSD and transfers it to host memory.

File Size	Write Prog	Write Meta	Exec Prog	Read Result
1.00 KB	0.038 ms	0.007 ms	0.243 ms	0.012 ms
10.00 KB	0.007 ms	0.009 ms	0.151 ms	0.024 ms
100.00 KB	0.007 ms	0.005 ms	0.421 ms	0.095 ms
1.00 MB	0.012 ms	0.006 ms	1.832 ms	0.649 ms
10.00 MB	0.015 ms	0.008 ms	10.453 ms	8.162 ms
100.00 MB	0.046 ms	0.016 ms	95.838 ms	73.500 ms

TABLE 9.4: The runtime of offloading a program to Delilah with SCI enabled that reads a file from the underlying SSD and transfers it to host memory.

Selective Cache Invalidation (*SCI*) is currently the most efficient way to enable CPU caches in Delilah without encountering cache coherency issues. Suppose the CPU cache is enabled and SCI is not being utilised in Delilah. In that case, there is a chance of accessing stale data, both on the host and in the processing unit of the Daisy board, due to the lack of cache coherency protocols across the PCIe link. With SCI, the host application describes what pieces of the memory must be guaranteed to be up to date for either the internal device CPU or the host [23]. We apply the term *invalidation* for host-to-device invalidation, where the CPU cache will have pieces of memory invalidated, and we apply the term *flushing* for device-to-host flushing, where the CPU cache will flush changes to main memory.

SCI is implemented in practice as four 64-bit unsigned integers sent alongside the execution command. These integers denote how much of the data slot will be invalidated and flushed, respectively, and what the invalidation and flushing offset within the buffers are. Imagine an experiment where we offload a program that reads a single file from the underlying SSD to Delilah, after which the host reads it back to host memory. The host could transfer the file metadata, including path and size, to the data slot's beginning. The host application would then set the SCI invalidation size to the size of the metadata struct and an invalidation offset of zero. The flushing offset would be the size of the metadata struct, and the flushing size would be the read file size. These parameters would guarantee that Delilah is not accessing stale metadata and that the host will not read a stale file.

To show the importance of SCI, we have recreated the experiment above. Table 9.3 shows the resulting performance without SCI enabled. Without SCI, Delilah has to guarantee cache coherency of the whole buffer, thus spending time invalidating and flushing untouched parts. In Table 9.4, we see the same experiment, but with SCI configured to invalidate and flush only the bare minimum for the experiment to return valid data. This experiment is made with data slots with a size of 1 gigabyte. It should be noted that when we read less than a megabyte, correctly configured SCI values improve the program execution time by a factor of 1,000.

Of course, since the speedups of SCI come from Delilah being strategic about which parts of the buffer to invalidate and flush, the efficiency of SCI scales inversely with the usage of the buffer. Reading a 100 MB file from SSD to Delilah's memory takes 85 ms. Invalidating or flushing the entirety of a 1 GB file takes approximately 115 ms. Without SCI, the execution can be estimated to take 115 ms (1 GB invalidation) + 115 ms (1 GB flushing) + 85 ms (100 MB reading) = 315 ms. These numbers perfectly match the numbers in Table 9.3. However, with SCI, we can make this 0 ms (< 1 KB invalidation) + 12 ms (100 MB flushing) + 85 (100 MB reading) = 97 ms, which perfectly matches the numbers seen in Table 9.4.

Finally, it should be noted that the alternatives to SCI are limited. Placing data slots in a memory region connected via a Cache Coherent Interconnect (CCI) gives automatic cache coherence. However, on the Daisy OpenSSD, only the PS memory, which has limited capacity, is placed behind a Cache Coherent Interconnect. Another alternative is to avoid entirely making use of the CPU cache and always access memory directly. This approach, in turn, simplifies memory management but introduces the costly access latency of over one hundred nanoseconds. Therefore, these alternatives are infeasible in practice, and SCI is considered the only solution to managing cache coherency.

In conclusion, SCI is a critical feature of Delilah, enabling host applications to describe the memory access patterns, thus maximising the efficiency of Delilah's cache coherence management.

9.6 Block Design

The block design of Delilah is a continuation and optimised version of the `Daisy_M.2_PCIe_MIG_201901_20210413` block design by CRZ Technology [11]. This example block design is centralised around a Xilinx UltraScale+ IP, enabling processing in the PS domain. From the UltraScale+ IP, there are connections to two MIGs, two XDMA IPs connecting to NVMe, one XDMA connecting to the host via PCIe, and multiple support IPs, including Utility Vector Logics, Processor System Resets, AXI Interconnects, VIOs (Virtual Input/Output), and AXI GPIOs.

Delilah's block design is different from the example in several ways. First, we remove one of the XDMA IPs connecting to the underlying SSDs. We do

this as Delilah's experimentation has focused on setups with a single SSD. The XDMA IPs in the block design are some of the most resource-consuming IPs, so removing an unused IP frees up significant capacity on the FPGA, decreasing negative slack on the design. Delilah's block design has only two XDMA IPs: one for connecting to the host and one for connecting to the underlying SSD.

Another significant change is the removal of one of the DDR MIGs. While both the board and the block design can accommodate two external RAM modules, Petalinux 2019.1 only supports two memory regions at a time. One of the memory regions is the internal PS memory used for the operating system, while the other is one of the two DDR MIGs. It is possible to map both MIGs, given that their physical address space is merged into a single contiguous region. However, we removed one of the MIGs due to the risk of issues with a data slot mapped into two different RAM sticks, since the extra capacity is not needed for the current experimental setup. As with the XDMA IP, this also frees up space, which can improve the performance by reducing negative slack.

In Delilah's block design, we have added four HLS IPs to filter over a buffer more efficiently. In Section 9.7, we will describe these IPs in further detail. It should be noted that the HLS IPs run on a clock domain with a higher frequency to improve their throughput.

Besides changes to the IP, in the form of changes to the AXI Interconnects, we have also added an extra GPIO IP to send interrupt signals to the host. Delilah's handling of interrupts is documented in Section 9.4.1.

9.6.1 Lessons Learnt

Creating this block design has left us with several lessons learnt, which will be outlined in this subsection.

- **Real Estate is a Key Challenge**: Every FPGA has a limited amount of Flip Flops (FFs) and Look Up Tables (LUTs). However, utilising these resources does not necessarily scale linearly with the amount of IPs. This unexpected scaling occurs since IPs must often be placed near each other. This placement requirement means that even though there might be a significant amount of FFs and LUTs available on the FPGA, the block design can fail implementation or have a critical amount of congestion, because of the inability to place related IPs near each other. This limitation is also why we have removed several IPs from the original CRZ block design. We did this to avoid congestion on the board, which would have caused performance issues and prevented us from using other IPs like High-Level Synthesis.
- **Slowest Clock Frequency Determines Throughput**: While experimenting with the block design, we learnt that the slowest clock domain always determines the throughput of a given data path or component. This characteristic can lead to an unexpected

phenomenon where the same operation performs differently depending on the source of the operation. For example, accessing a given block of data in a data slot is faster from the host than the device's processing unit. This performance characteristic is the case because the clock frequencies of the MIG and XDMA IPs are high. In contrast, the clock frequency of the MPSoC is low, artificially limiting the throughput of data access to less than what the MIG can handle.

- **Version Control with Git**: Utilising version control systems such as Git is critical for managing the complexity of block designs. Block designs are complex and often require reverting to previous states after unsuccessful experimentation, which is not trivial to perform manually. Git enables tracking changes and easy rollback to stable versions. However, version control with FPGAs is challenging due to the extensive state and numerous files involved. Properly setting up version control requires careful planning to manage the large number of files generated by FPGA toolchains and track all relevant states and configurations.

9.7 Hardware-accelerated Filtering

By implementing our computational storage processor on an FPGA architecture, we can experiment with High-Level Synthesis (HLS) and specialised accelerators. Filtering is chosen as a natural starting point due to its general applicability.

Our filtering accelerator supports five modes: equality, inequality, less than or equal, greater than or equal, and between-inclusive.

```
typedef uint32_t filter_t;

uint32_t filter(filter_t *in, filter_t *out, uint32_t num,
                uint8_t op, filter_t comp1, filter_t comp2) {
#pragma HLS INTERFACE m_axi port=in offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=num bundle=control
#pragma HLS INTERFACE s_axilite port=op bundle=control
#pragma HLS INTERFACE s_axilite port=comp1 bundle=control
#pragma HLS INTERFACE s_axilite port=comp2 bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

  ...

}
```

LISTING 9.10: The signature of our hardware accelerated filtering operator.

Listing 9.10 shows the signature of our hardware-accelerated operator. The function accepts two pointers (one for input and one for output), an opcode to select the mode, and two comparison values. The `#pragma` directives specify the use of two AXI ports: a high-throughput AXI port for data access and a low-latency AXI Lite port for reading and writing function parameters. The

`bundle` directive instructs the HLS compiler to merge the signals into the two AXI ports instead of creating seven individual ports for each variable.

```
uint8_t k = 0;  // Vector result counter
uint32_t c = 0, // Overall result counter
    i,          // Vector iterator
    j;          // Value iterator

filter_t in_buf[BUF_SIZE];
filter_t out_buf[BUF_SIZE];

switch (op) {
  case DELILAH_FILTER_EQ:
  Vector_Loop_EQ:
    for (i = 0; i < num; i++) {
      memcpy(in_buf, &in[i * BUF_SIZE], BUF_SIZE * sizeof(filter_t)
    );
    Value_Loop_EQ:
      for (j = 0; j < BUF_SIZE; j++) {
#pragma HLS UNROLL factor = 1
#pragma HLS PIPELINE
        if (in_buf[j] == comp1) {
          out_buf[k] = BUF_SIZE * i + j;
          k += 1;
        }
      }
      memcpy(out + c, out_buf, k * sizeof(filter_t));
      c += k;
      k = 0;
    }
    break;

    ...
}
```

LISTING 9.11: The implementation of one of the operations of our hardware accelerated filtering operator.

Listing 9.11 shows the implementation of the filtering operator in equality mode. The operator utilises vectorisation, processing sets of 256 elements at once. This approach minimises individual sequential memory requests for 32-bit elements, instead issuing 1-kilobyte memory reads and writes. The vector is stored in local BRAM within the accelerator.

During execution, the operator first iterates over the number of vectors. Each vector iterates over the elements in a pipelined manner, where it begins to compare the next element before the current one finishes. The vector results counter increments on finding a match, and the index is written to the output buffer. The output buffer is then written back to the output pointer for each vector.

The accelerator is managed and invoked using the Hardware module of Delilah, described in Section 9.4.1.

9.7.1 Lessons Learnt

Creating this hardware-accelerated filtering IP has left us with several lessons learnt, which will be outlined in this subsection.

- **Optimisations are not Automatically Inferable**: One of the major challenges we faced when developing the filtering IP was memory latency. The HLS toolchain provides many ways to optimise data throughput in loops, such as loop unrolling, pipelining, and array partitioning. However, these optimisations may be overly costly to implement without providing tangible benefits. In our case, almost the entirety of the time was spent waiting for memory accesses, so we saw a major speedup by manually bundling accesses together via memory copies to buffers. At the same time, we saw no tangible increase in performance with loop unrolling. Another challenge is the inference of loop boundaries. For example, given a loop of up to 1024 iterations counted using a 16-bit integer, the compiler may infer that the loop must be able to handle up to 65536 iterations due to the 16-bit counter, even if the loop never has more than 1024 iterations. In this case, explicitly adding code to exit the loop at 1024 iterations can be necessary, so that the compiler does not optimise for irrelevant cases. This explicit addition helps the compiler understand the loop, even if the code is never triggered.
- **Optimisations Require Real-Estate**: On processing units, we are used to loop unrolling as a way to execute several iterations of the loop serially, i.e., the overhead of comparing loop counters is incurred less frequently, and proportionally more time is spent in the loop body. However, the characteristics of loop unrolling in the context of FPGAs are different. Here, loop unrolling means that the unrolled loop iterations happen concurrently. As such, it can be expected that loop unrolling to the second degree will halve the number of iterations but double the amount of real estate needed for the loop. At the same time, stateful variables accessed in the loop are now prone to race conditions. Due to this, loop unrolling may not always be beneficial.
- **Debugging Tools are Limited**: While developing the filtering IP, we experienced several different issues with the IP not performing as expected. For example, we encountered a condition where the IP would iterate over a given array but could never access every other array entry. The code passed testing on the development machine, and the toolchain did not indicate any problems. Ultimately, the code and the IP itself were demonstrated to be correctly written and configured. However, an automatically inferred flag in the block design forced all memory accesses to be 64-bit aligned, while the array contained 32-bit elements. The toolchain does not detect issues like these, and no debugging tools exist to find them. Developing HLS IPs is a challenging task requiring attention to detail and creativity in debugging.

9.8 Conclusion

The implementation of Delilah is based on the requirements and design decisions from previous chapters. The Delilah driver is based on the Eid-Hermes driver but has several extensions, including the use of io_uring instead of traditional filesystem calls. Delilah uses the `uring_cmd` function to manage DMA requests or forward commands directly to the device. Commands are processed parallel across two layers of queues, inner and outer. Eight specific commands are implemented, including program execution, DMA transfers, cache clearing, and device state management.

The Device Controller resides in Petalinux within the Xilinx MPSoCs Processing System. The device controller has several modules.

1. **Command Module**: Defines and handles supported commands and updates execution status in control registers.
2. **Config Module**: Stores adjustable configuration parameters like physical slot locations and engine count.
3. **Functions Module**: Maps eBPF call instructions to executable code using a function table.
4. **Hermes Module**: Deprecated. The module structures commands, requests, and responses for compatibility with the old Eid-Hermes protocol.
5. **Hardware Module**: Manages hardware-accelerated functions and integrates with High-Level Synthesis drivers for execution.
6. **Interrupt Module**: Utilises GPIO to signal command completion to the host.
7. **Loader Module**: Configures and manages the execution environment, including setting up uBPF VMs for executing eBPF programs.
8. **Memory Module**: Manages memory regions for data and program slots and ensures cache coherency.
9. **Utilities Module**: Provides convenience functions and macros for error handling, time measurement, memory size definitions, and logging.
10. **TSL Module**: Experimental. Integrates Delilah with the Template SIMD Library (TSL) for SIMD operations, potentially improving performance for specific workloads.

Insights from previous studies revealed suboptimal memory access latency on the Xilinx ZCU102 chip. Due to this, Delilah employs CPU caching to improve data processing speed for sequential accesses. Selective Cache Invalidation (SCI) within the memory module is implemented to enable CPU caching efficiently without encountering cache coherency issues. Delilah

only guarantees cache coherence on a subset of memory buffers by allowing the host application to describe memory access patterns with SCI, thus decreasing time spent on managing memory.

In addition to cache optimisation, Delilah's block design is an optimised version of the `Daisy_M.2_PCIe_MIG_201901_20210413` design by CRZ Technology. Modifications include removing redundant XDMA IPs and DDR MIGs, which frees up FPGA capacity and reduces negative slack. The block design is extended with hardware-accelerated filtering IPs that run on separate clock domains with higher frequencies. Lastly, a GPIO IP for interrupt signalling to the host is added.

The Petalinux SDK by Xilinx is used to deploy Delilah. Delilah is added as a user-space application recipe that can be built with either Release or Debug mode BitBake definitions. Furthermore, modifications to the device tree ensure guaranteed memory allocation for UDMA and expose hardware-accelerated filtering engines via user-space IO (UIO).

In conclusion, the implementation of Delilah fits the requirements and decisions previously described. The implementation, including all underlying modules, efficiently supports the workload of integrated data analysis pipelines. More specifically, the implementation gives enough control to the host application to allow for significant flexibility. It allows the host to schedule program execution without adapting to a custom scheduler within Delilah. On top of this, the protocol and modules are extensible if further work shows that particular functionality could improve the integration with host-side applications or integrated data analysis pipelines.

Chapter 10

Summary

In the first two chapters of Part II, we have outlined five critical requirements for a novel computational storage processor based on the findings of Part I.

We outlined how memory management plays a critical role in designing and implementing a computational storage device integrating with host-side integrated data pipelines. Integrated data pipelines are efficient because the memory layout is consistent across many operations and does not require memory to be adapted to fit the constraints of a single operation. For computational storage to be practical for integrated data pipelines, the memory layout of the computational storage device must also be consistent. We proposed splitting memory into four categories: control registers, program slots, data slots, and shared data slots. These regions are static in size and location to simplify the integration, as the host-side pipelines can expect data never to move around unexpectedly. However, memory management in computational storage is still a challenge only partially solved. Having separately managed memory on the device poses several challenges concerning cache coherence and fragmentation. Selective Cache Invalidation, described in Section 9.5, solves the immediate challenges of cache coherence, but further work is necessary to ensure efficiency on larger workloads. Fragmentation is an unsolved problem caused by the static size of the slots. Applications are restricted to data slots of 1 GB, regardless of whether the application needs this capacity. Ultimately, we learnt that having static memory regions is favourable for integration with the host but does induce challenges like fragmentation, which would not occur with dynamically allocated regions.

We identified executing eBPF as essential to building a novel computational storage device. eBPF was selected because it is a vendor-neutral, lightweight ISA. However, relying on eBPF is not without challenges. eBPF has constraints on supported operations and lacks instructions to express floating point arithmetic and file I/Os. As such, the `call imm` instruction is needed to invoke device-specific functionality known neither by the host nor the eBPF program. We call this eBPF sidestepping since the `call imm` instruction interrupts execution and hands control over to Delilah, which then conducts the inexpressible operation. While the solution of sidestepping resolves the challenges of executing inexpressible functions, further work is still needed on enumeration and identifying the callable functions. In Eid-Hermes, the

SNIA standard, and the NVMe standard, no such thing as sidestepping is defined, and it is unclear whether the `call imm` instruction is used at all. In Delilah, the instruction is used, but there is no way for the host to identify the functionality accessible via the instruction. One must know the target device and version to determine which functions are invokable and with what signature.

Delilah is based on the Daisy OpenSSD and Eid-Hermes. While these were sufficient to support the development of Delilah, several challenges were identified. Eid-Hermes relied on a synchronous interface and a protocol with semantically invalid states. These had to be alleviated, as a synchronous interface is highly impractical for high-performance systems. The Daisy OpenSSD proved to be complicated and challenging to utilise efficiently. In Section 11.2, we show in further detail that the Daisy OpenSSD is impractical for specific access patterns due to very high memory latencies. Further work is necessary to determine the feasibility of the Daisy OpenSSD to support computational storage devices like Delilah, and whether the Daisy+ OpenSSD or boards from alternative vendors would serve as a more suitable hardware platform.

The fact that Eid-Hermes proved to be insufficient on its own, and that the standards of SNIA and NVMe have certain limitations, for example, with regards to cache coherence and the eBPF `call imm` instruction, indicates that we are undergoing a paradigm shift in computational storage. Only a few years ago, the interfaces of computational storage devices were generally white-box. While the implementations may be proprietary, the interfaces were always open, and the underlying operations and side effects were known. With the emergence of eBPF and true vendor neutrality, questions arise about the portability of some operations. For example, since functions are only identified by their signature, vendors may implement functions that appear to do the same but may have very different underlying instructions or side effects.

While we consider the design and implementation of Delilah to be successful, it is also evident that it is has limitations. While designing the block design, we saw that clock frequencies of a single IP can undermine the entire system's performance, and that real estate on the FPGA can cause unwanted congestion or performance issues. After designing and implementing Delilah, we are left with unresolved questions on how to guarantee efficiency on such a diverse system, spanning across a driver, a controller, a block design, and an operating system. How can we benchmark Delilah's system stack and identify bottlenecks?

Part III

Evaluation

Chapter 11

Memory Access Evaluation

To determine the efficiency of Delilah, we must first experiment to determine if and where any memory bottlenecks exist on the device. Note that we have already shown the effects of cache coherency on the Delilah architecture in Section 9.5.

11.1 DMA vs. CMB

In the design of Delilah, outlined in Chapter 8, we determined that the control registers should be placed in low-latency memory regions and accessed via Controller Memory Buffers (CMB). We also determined that data, often more significant in size, should be accessed via Direct Memory Access (DMA). This design decision follows the design of Eid-Hermes, outlined in Section 3.3.10.

We have experimented with the latency of data access on the device via Direct Memory Access and Controller Memory Buffers. For the CMB experiments, data is read and written to a reserved area in the BAR0 configuration region. For DMA, we read and write to a data slot on the device.

Table 11.1 shows the latency of writing from 1 byte and up to 10 megabytes in increments of one order of magnitude. The experiment opens

Buffer Size	Read		Write	
1.00 B	0.001 ms	1.000 MB/s	0.000 ms	inf MB/s
10.00 B	0.002 ms	5.000 MB/s	0.000 ms	inf MB/s
100.00 B	0.012 ms	8.333 MB/s	0.000 ms	inf MB/s
1.00 KB	0.103 ms	9.709 MB/s	0.000 ms	inf MB/s
10.00 KB	8.257 ms	1.211 MB/s	2.013 ms	4.968 MB/s
100.00 KB	82.490 ms	1.212 MB/s	20.283 ms	4.930 MB/s
1.00 MB	94.509 ms	10.581 MB/s	49.742 ms	20.104 MB/s
10.00 MB	945.134 ms	10.581 MB/s	498.245 ms	20.070 MB/s

TABLE 11.1: The latency of writing various amounts of data from the host to data slots via Controller Memory Buffers (CMB).

Buffer Size	Read		Write	
1.00 B	0.007 ms	0.143 MB/s	0.030 ms	0.033 MB/s
10.00 B	0.015 ms	0.667 MB/s	0.006 ms	1.667 MB/s
100.00 B	0.016 ms	6.250 MB/s	0.006 ms	16.667 MB/s
1.00 KB	0.007 ms	142.857 MB/s	0.020 ms	50.000 MB/s
10.00 KB	0.011 ms	909.091 MB/s	0.033 ms	303.030 MB/s
100.00 KB	0.064 ms	1562.500 MB/s	0.079 ms	1265.823 MB/s
1.00 MB	0.600 ms	1666.667 MB/s	0.733 ms	1364.256 MB/s
10.00 MB	6.419 ms	1557.875 MB/s	7.489 ms	1335.292 MB/s
100.00 MB	58.557 ms	1707.738 MB/s	74.598 ms	1340.519 MB/s
1.00 GB	614.847 ms	1626.421 MB/s	737.469 ms	1355.989 MB/s

TABLE 11.2: The latency of writing various amounts of data from the host to data slots via Direct Memory Access (DMA).

Buffer Size	Read		Write	
1.00 B	0.041 ms	0.098 MB/s	0.041 ms	0.098 MB/s
10.00 B	0.020 ms	2.000 MB/s	0.023 ms	1.739 MB/s
100.00 B	0.029 ms	13.793 MB/s	0.023 ms	17.391 MB/s
1.00 KB	0.021 ms	190.476 MB/s	0.033 ms	121.212 MB/s
10.00 KB	0.037 ms	1081.081 MB/s	0.048 ms	833.333 MB/s
100.00 KB	0.218 ms	1834.862 MB/s	0.296 ms	1351.351 MB/s
1.00 MB	1.617 ms	2473.717 MB/s	2.632 ms	1519.757 MB/s
10.00 MB	15.119 ms	2645.678 MB/s	24.106 ms	1659.338 MB/s
100.00 MB	168.119 ms	2379.267 MB/s	254.664 ms	1570.697 MB/s
1.00 GB	1809.847 ms	2210.132 MB/s	2704.805 ms	1478.850 MB/s

TABLE 11.3: The latency of writing various amounts of data from the host to data slots via Direct Memory Access (DMA) using all four H2C/C2H channels.

the BAR0 configuration register used by Delilah to hold command requests and responses. The region is open by memory-mapping `/sys/bus/pci/devices/0000:01:00.0/resource0`, which is the 0th Delilah device, and the 0th resource, which is BAR0. The memory map is opened using `O_SYNC`.

We can observe from the results that the transfer is almost instantaneous for small transfers in the order of bytes. For 1 KB and below, the reads take up to one-tenth of a millisecond, and the writes are too fast to measure. This pattern could be caused by write-back strategies by the device, where the write completes when the PCIe bus has received the write, and not when it is written to the device memory.

For transfers larger than 1 KB, the efficiency decreases. For 10 KB and 100 KB, we see a decrease in throughput in the factor of an order of magnitude. For 1 MB and 10 MB, the efficiency is similar to < 1 KB for throughput, but with a linear scaling.

The results differ significantly from those of Direct Memory Access transfers, observed in Table 11.2. We can observe a high minimum latency on all transfers. Whereas CMB has a lower bound of transfer of 0.001 ms, the lower bound of DMA is indeterministic and varies between 0.007 ms and 0.016 ms. However, for transfers above 1 KB, the efficiency spikes at orders of magnitude compared to CMB. For example, a transfer of 1 KB takes 0.103 ms via CMB and 0.007 ms via DMA, with a throughput of 9.709 MB/s and 142.857 MB/s, respectively. For 10 MB, the throughput is 10.581 MB/s and 2645.678 MB/s, respectively.

This performance characteristic shows that CMB is optimal for small transfers, for example, requests and responses of less than 1 KB. However, offloading the transfer to the DMA engine for larger transfers seems optimal.

In Table 11.3, we see the same DMA experiment but transferring four times as much data via four H2C/C2H channels. In this experiment, we observe how using multiple channels increases the throughput, but not at a factor of four. We see an almost 1000 MB/s improvement in throughput for reads, whereas we only see around an increase of 100-200 MB/s in throughput on writes. This trend strongly indicates that either the DMA channels are not isolated and share the same PCIe lanes, or the underlying AXI connection from XDMA to the DDR4 MIGs is exhausted. However, in conclusion, since the performance of using multiple DMA channels is never less than using just one, it is better to use multiple channels where possible.

11.2 Memory Accesses

As outlined in Chapter 4, the Daisy OpenSSD has an internal memory, denoted PS memory, and can have up to two external DDR4 DRAM sticks attached. In this experiment, we want to see the different performance characteristics. We have mounted two `M393A2K40BB1-CRC` memory sticks from

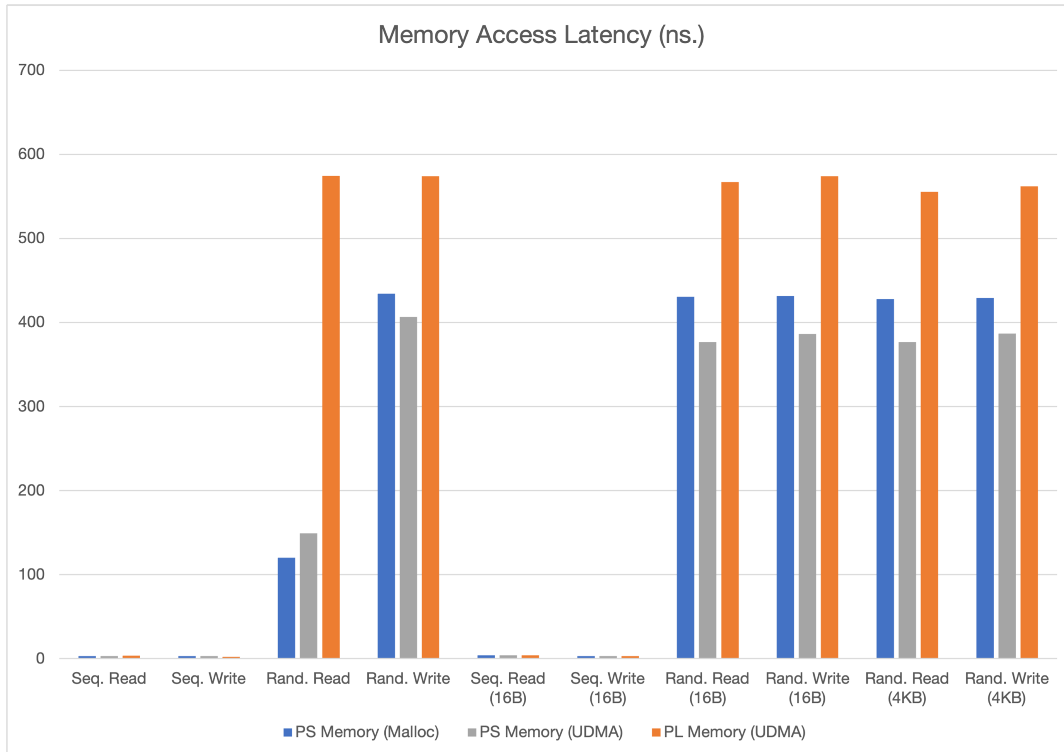


FIGURE 11.1: Average latency of a single memory access when running 32 million accesses in a row. The results are shown in linear scaling. *Seq* denotes sequential, *rand* denotes random, and the values in parenthesis denotes the alignment.

Samsung in the PL DIMM slots. These memory sticks have a frequency of 2400 MHz, a size of 16 GB, and a component composition of (2G x 4) x 18 [35]. According to its user guide, the Daisy OpenSSD comes with one component of `MT53B768M32D4NQ-062` LPDDR [12]. This memory component has a frequency of 1600 MHz, a size of 2 GB, and a component composition of 768M x 32 [37].

The experiment to benchmark and compare the two memory components is straightforward. We design a set of registered functions (see Section 9.4.1) that execute 32 million memory loads and stores. We experiment with reads and writes, sequential and random accesses, and with and without 16-byte and 4-kilobyte alignment. Furthermore, we access PS memory, both using UDMA and via a region allocated via malloc, to try to identify the performance characteristics of UDMA. The experiment yields an amortised access latency, measured in nanoseconds.

The results of the experiments are illustrated in Figures 11.1 and 11.2. The first notable observation is the massive contrast between sequential and random accesses. Sequential accesses perform at near-zero latency, regardless of alignment. This performance characteristic likely stems from the efficient operation of predictive cache prefetching mechanisms and the low latency of approximately three nanoseconds of the L1/L2 caches. Furthermore, we suspect aligned writes are temporarily cached in L1/L2 caches, followed by

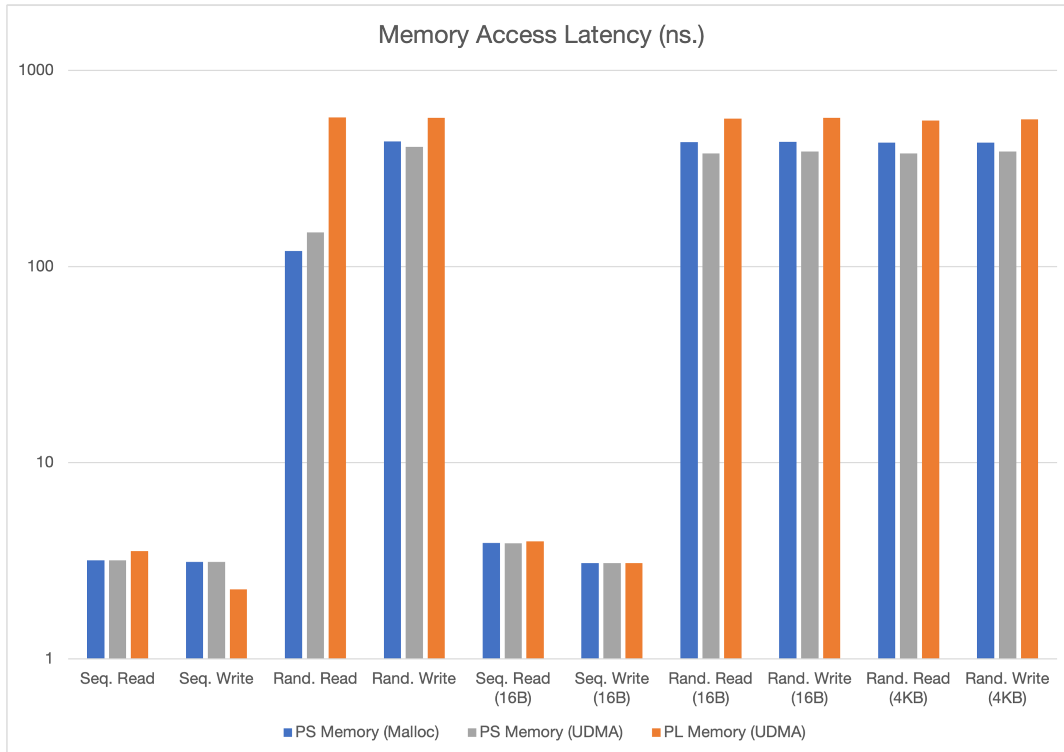


FIGURE 11.2: Average latency of a single memory access when running 32 million accesses in a row. The results are shown in logarithmic scaling. *Seq* denotes sequential, *rand* denotes random, and the values in parenthesis denotes the alignment.

combined memory writes, potentially leading to increased bus utilisation.

Notably, the choice of an underlying RAM module for sequential reads and writes appears inconsequential, as the caching efficiency of L1/L2 negates any significant impact. On the other hand, random accesses show a considerable latency increase, with access times around two orders of magnitude slower than sequential accesses. This performance bottleneck likely comes from the inability of predictive prefetching to cache reads ahead of time, thus always causing a cache miss. Meanwhile, any caching of writes with the intention of write combining now fails.

We observe an outlier in random reads to PS memory, where access latencies are 3–4 times better than random writes and aligned random accesses. The cause of this latency discrepancy remains an open question, but it is reproducible between runs with high precision.

Furthermore, experiments exploring the alignment of random accesses show that alignment has minimal effect on memory access latencies, indicating that other factors, like choice of memory module and distance from CPU, predominantly impact access speed in such scenarios.

Access to memory via UDMA does not increase access latency, as one would expect when adding another layer of abstraction. Instead, it reduces latency

in most cases. The reason for this is likely different CPU cache implementations. According to the README of the library, Linux will turn off CPU caching for memory-mapped buffers on architectures like ARM and ARM64, where cache aliasing problems can occur [26]. The framework offers an alternative path to enable CPU caches, which may be the difference in access performance we observe.

We can compare our results to Bansal et al., who have conducted experiments on the memory subsystem of a Xilinx Ultrascale+ MPSoC ZCU102 [4]. The ZCU102 MPSoC is the same as the one on the Daisy OpenSSD. However, it should be noted that they have different memory modules, namely the Kingston `KVR21SE15S8` in PS and Micron `MT40A256M16GE-075E` in PL, which only enable us to compare the tendencies and not the specific latencies.

Their experimentation shows an L1 latency of 3 nanoseconds, identical to ours. Furthermore, they show an L2 latency of 20 nanoseconds. Most importantly, they show that the latency increases significantly when working sets become more extensive than what can feasibly be cached in L1 and L2. Their experimentation shows that randomised reads and writes are in the order of hundreds of nanoseconds, similar to our experiments.

To conclude, experiments show the importance of locality when offloading to Delilah. If the offloaded programs exhibit random access patterns, Delilah will likely be unable to outperform the host or other accelerators. Furthermore, Delilah's memory management architecture seems optimal for the constraints of the Daisy OpenSSD.

Accessing the slower PS memory tends to come with lower latency, even though the PL DDR4 DRAM sticks have a higher frequency than PS. This characteristic likely stems from inefficiencies in PS-to-PL ports or AXI connections from the CPU to the MIGs. This hypothesis is probable because other researchers have seen a similar memory access latency on the Xilinx UltraScale+ MPSoC.

Since access latency is highly dependent on whether data is accessed sequentially or randomly, the offloaded workload plays a critical role in the throughput. Sequential workloads, such as filtering and transformations, will be efficient on Delilah due to their sequential access patterns. Sequential data processing ensures that data is fetched contiguously, reducing cache misses and utilising the bandwidth of the memory components. This characteristic makes Delilah well-suited for applications involving tasks that require reading and writing data in a predictable order.

On the other hand, workloads characterised by random access patterns, such as sorting, will exhibit significantly lower throughput. In these workloads, there will be frequent cache misses and an inability to anticipate the following memory access, causing prefetching to miss continuously. Sorting algorithms, which often involve frequent random reads and writes, are inhibited by the increased latency associated with random memory accesses. This latency bottleneck reduces throughput, as the memory components spend

more time addressing cache misses and retrieving data from slower memory layers. Thus, while Delilah can achieve high throughput for sequential access patterns, it will face performance challenges with tasks requiring random data access.

11.3 Conclusion

In this chapter, we have conducted two experiments to understand the memory access characteristics of the Daisy OpenSSD. Exploring Direct Memory Access (DMA) versus Controller Memory Buffers (CMB) reveal that while CMB is highly efficient for small transfers, DMA significantly outperforms CMB for more extensive data transfers, especially when utilising multiple channels.

Our memory access experiments show the role of access patterns. Sequential accesses benefit greatly from cache prefetching, resulting in minimal latency, whereas random accesses suffer from significant latency increases due to frequent cache misses. The observed differences in memory access latencies between PS and PL memories may be caused by underlying inefficiencies in the PS-to-PL ports or AXI connections despite the higher frequency of the PL DDR4 DRAM sticks.

Our findings on the inefficiencies of memory accesses match existing research, such as the work of Bansal et al., particularly regarding the latency penalties associated with random access patterns. Similar trends were observed in latency increases for random accesses, and the effectiveness of L1/L2 caches further validated our results.

Our exploration and experimentation show the importance of understanding memory access patterns to optimise the use of Delilah. Delilah may not provide a significant performance advantage for workloads with random accesses over the host or other accelerators. However, Delilah's memory management architecture is highly efficient for tasks with predictable access patterns.

Now, the question is how problematic these access latencies are in reality. In the next chapter, we will conduct an experiment using a SSB query to determine the viability of Delilah with a real-world use case. This experiment will help us understand the practical implications of memory access latencies, and Delilah's performance in addressing complex queries and data processing tasks typical in database operations.

Chapter 12

Experimental Evaluation

While running micro-benchmarks and experimenting with subcomponents of the Delilah system can reveal potential bottlenecks, some performance characteristics only show when components are active simultaneously.

Alongside Alexander Krause and Johannes Pietrzyk from TU Dresden's Database Research Group, we have designed and implemented a host-side application for experimenting with Star Schema Benchmark (SSB) query 3.3 [41]. The 3.3 query, seen in Listing 12.1, is interesting because it combines several typical database operations. It has conversion, filtering, and aggregation spread over multiple tables, requiring a join. These characteristics make it relevant to the requirements because it tests the ability to express complex queries in eBPF, with memory spanning multiple programs and operations. On top of this, it triggers both sequential and random access patterns to memory.

LISTING 12.1: Star Schema Benchmark Query 3.3.

```
SELECT c_city,
       s_city,
       d_year,
       Sum(lo_revenue) AS revenue
FROM   customer,
       lineorder,
       supplier,
       date
WHERE  lo_custkey = c_custkey
       AND lo_suppkey = s_suppkey
       AND lo_orderdate = d_datekey
       AND ( c_city = 'UNITED KI1'
             OR c_city = 'UNITED KI5' )
       AND ( s_city = 'UNITED KI1'
             OR s_city = 'UNITED KI5' )
       AND d_year >= 1992
       AND d_year <= 1997
GROUP  BY c_city,
          s_city,
          d_year
```

```
ORDER BY d_year ASC,
         revenue DESC;
```

12.1 Experimental Setup

The experimental framework uses a Delilah device connected to a host via PCIe3 x8. The host has an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with four cores and 32 GB DDR3. It runs Ubuntu 22.04 (Linux 6.2.6), and Clang 14.0 compiles eBPF programs.

Furthermore, the setup has a development server, which has a 12th Gen Intel(R) Core(TM) i7-12700KF with 12 cores and 32 GB DDR4, and it is running Ubuntu 16.04 (Linux 4.15.0). We use this server to compile the block design, offload FPGA bitstream and OS image to the Daisy platform, and monitor experiments via JTAG.

All experiments in this chapter have been repeated ten times, and all numbers are averages of the ten runs. Unless explicitly noted, Delilah runs in JIT mode, and the shared memory slot is placed in PL memory.

12.2 Experiment 1: Interpretation vs. Just-in-Time

In Figure 12.1, we see a significant difference between Delilah's performance in JIT mode versus interpreted mode, particularly when executing the SSB query fully on Delilah. Figure 12.2 shows a similar comparison, this time running the query on the host, thus only offloading data transfer from SSD to host in both JIT and interpreted modes.

Surprisingly, three out of four experiments show similar performance characteristics, an outlier being the interpreted mode for the full offload of the SSB 3.3 query. This observation indicates that for smaller programs with a limited set of instructions, the choice between JIT and interpreted mode may not significantly impact performance. In the host-only experiment, JIT runtime was measured to 5.59 seconds, only marginally beating the interpreted mode runtime of 5.60 seconds.

However, the difference in performance becomes much more notable when working with larger programs, showing an order-of-magnitude difference between JIT (runtime: 5.85 seconds) and interpreted mode (runtime: 53.77 seconds). The interpreted mode requires multiple ARM instructions for each eBPF instruction, as time and instructions are spent on parsing, decoding, and handling individual instructions.

This experimentation leads to a lesson learnt about the current version of Delilah and uBPF: computational storage processors are best suited to operate in JIT mode, with interpreted mode being reserved for smaller programs with minimal instructions. The risks of utilising interpreted mode cannot be

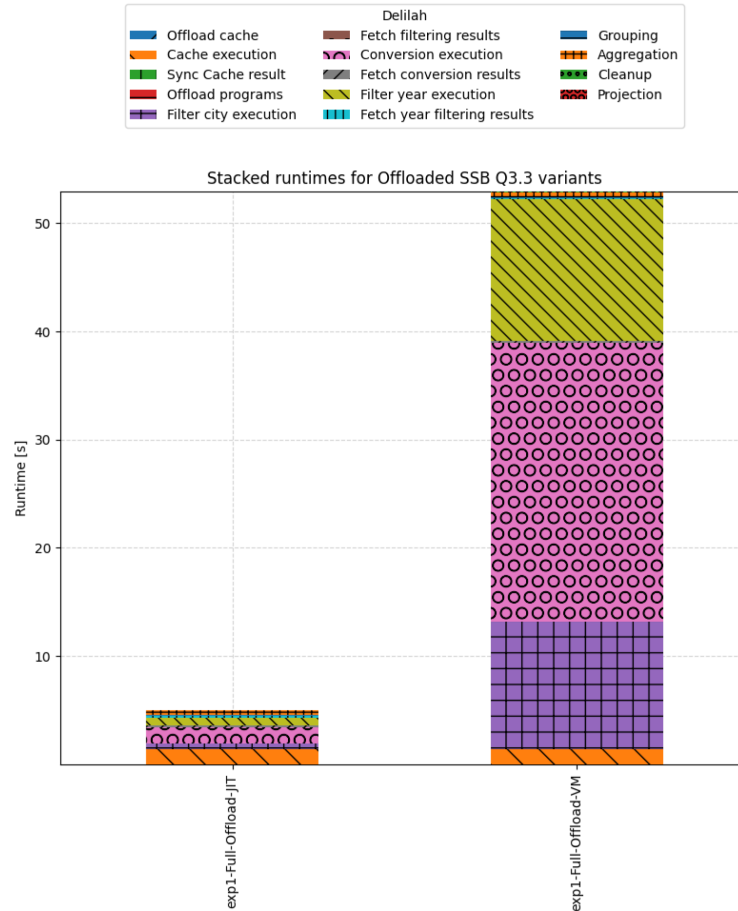


FIGURE 12.1: The result of a full offload of query 3.3, where we compare the runtime between running Delilah in JIT mode versus interpreted mode. In JIT mode, all query instructions are transpiled into ARM instructions native to the Daisy OpenSSD. In interpreted mode, an interpreter parses the instructions and executes them sequentially.

overstated, showing that JIT execution must be enabled for optimal performance. However, it should be noted that future versions of eBPF and uBPF may lead to different conclusions.

12.3 Experiment 2: Partitioned Execution

In Figure 12.3, we experiment with comparing the execution of operators that work with the maximum amount of data possible and operators that partition the data into smaller chunks. The partitioning aims to enable the host to process data earlier, instead of remaining idle until the operator has finished processing. We want to understand if a computational storage processor performs better when operators work on as much data as possible, or if partitioning increases performance by enabling other processing units, such as the host, to execute simultaneously. This experiment tests whether data pipelining is beneficial for computational storage.

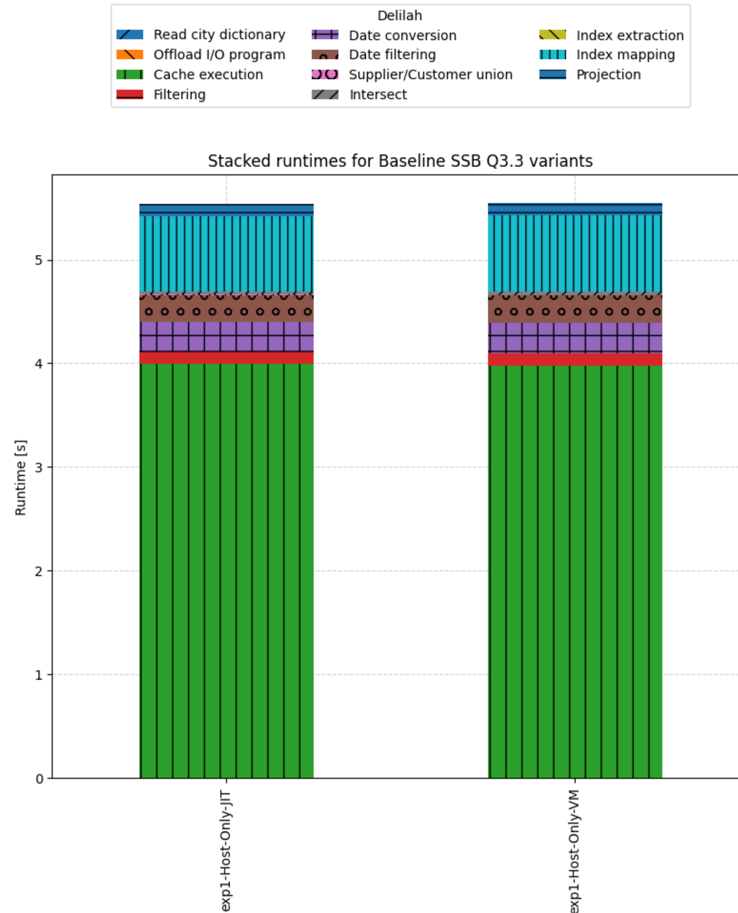


FIGURE 12.2: The result of the host-only experiment, where Delilah only fetches the files for the host without doing any processing. We compare the runtime between running Delilah in JIT mode versus interpreted mode. In JIT mode, all query instructions are transpiled into ARM instructions native to the Daisy OpenSSD. In interpreted mode, an interpreter parses the instructions and executes them sequentially.

The lessons learnt from this experiment are three-fold:

- When partitioning operators, using shared memory regions becomes a non-trivial task. In the unpartitioned scenario, approximately 1.5 seconds are spent reading all query columns, after which other operators can concurrently operate on the data. However, when only parts of a column are read intermittently, and the file size remains to be determined, managing and allocating shared memory regions raises significant challenges.
- Partitioning operators introduce communication and cache overhead. After processing segments of input data by an operator, the host is notified and must determine the next course of action, while the computational storage processor remains idle. Mechanisms such as Selective Cache Invalidation (SCI), discussed in Section 9.5, and JIT compilation,

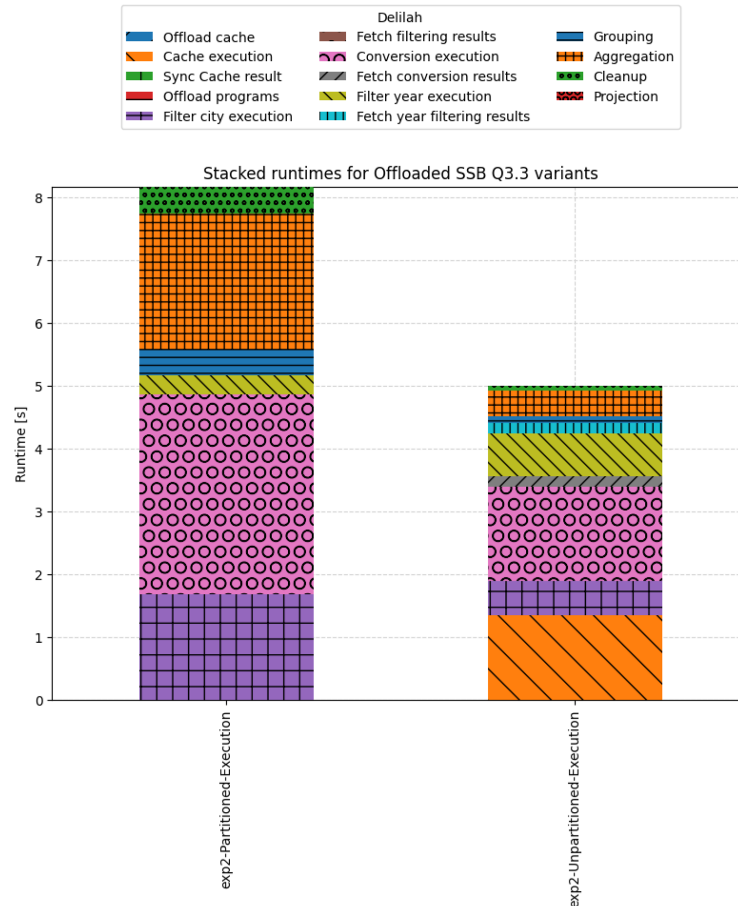


FIGURE 12.3: The result of running the full query on Delilah, either running to completion or in chunks of 128 MB.

are triggered per partition rather than per operation.

- Aggregation operators now incur significant overhead since the input to the aggregator comes in chunks. As such, the aggregator may need to make several passes over the same data.

It is most beneficial for computational storage processors to execute larger tasks and, if possible, avoid partitioning operators artificially.

12.4 Experiment 3: Filesystem Caching

In this experiment, we aim to understand the implications of utilising direct I/O for file reading by avoiding the traditional process of copying data from the filesystem cache to the user buffer. The result of this experiment can be seen in Figure 12.4.

As expected, the performance characteristics across all but one operator are identical, since the direct I/O only applies to a single operator. In this operator, we observe a difference of approximately 0.27 seconds, showing the benefits of utilising direct reads over traditional cached filesystem reads.

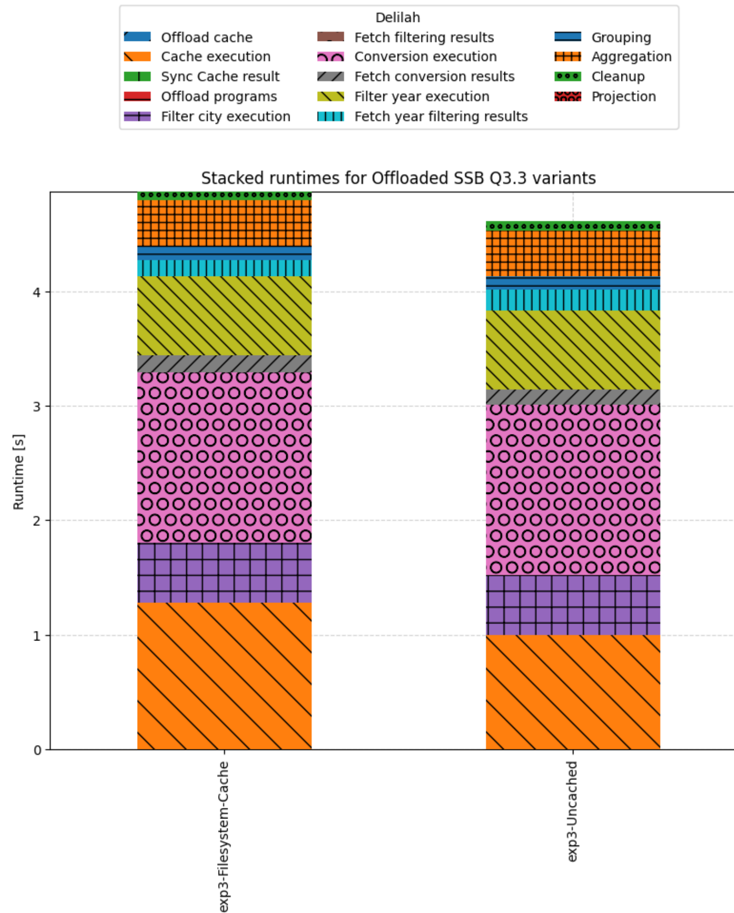


FIGURE 12.4: The result of running the full query on Delilah, either reading column files with or without the *O_DIRECT* flag. Contrary to other experiments, this experiment is executed with the shared memory slot in PS memory.

In conclusion, the experiment shows the importance of using direct I/O operations in computational storage processors. Using direct I/O where possible improves the performance of data retrieval and aligns better with the objective of computational storage processors: to improve performance in resource-constrained environments.

However, it should be noted that this experiment also introduces questions about the viability of design decisions. We observe that disabling filesystem caching via *O_DIRECT* improves performance by removing a level of indirection. However, *O_DIRECT* requires the memory region to be mapped into the kernel as standard memory. UDMA exposes the PL device memory as standard kernel-mapped memory, even though the memory is not initially mapped this way. Using UDMA introduces a level of indirection, making it incompatible with direct reads. This characteristic raises an unanswered open question about the design of computational storage devices: How can memory, which is not considered standard by the kernel, be efficiently utilised by the computational storage device? Is there an alternative UDMA that enables the computational storage device to map memory more

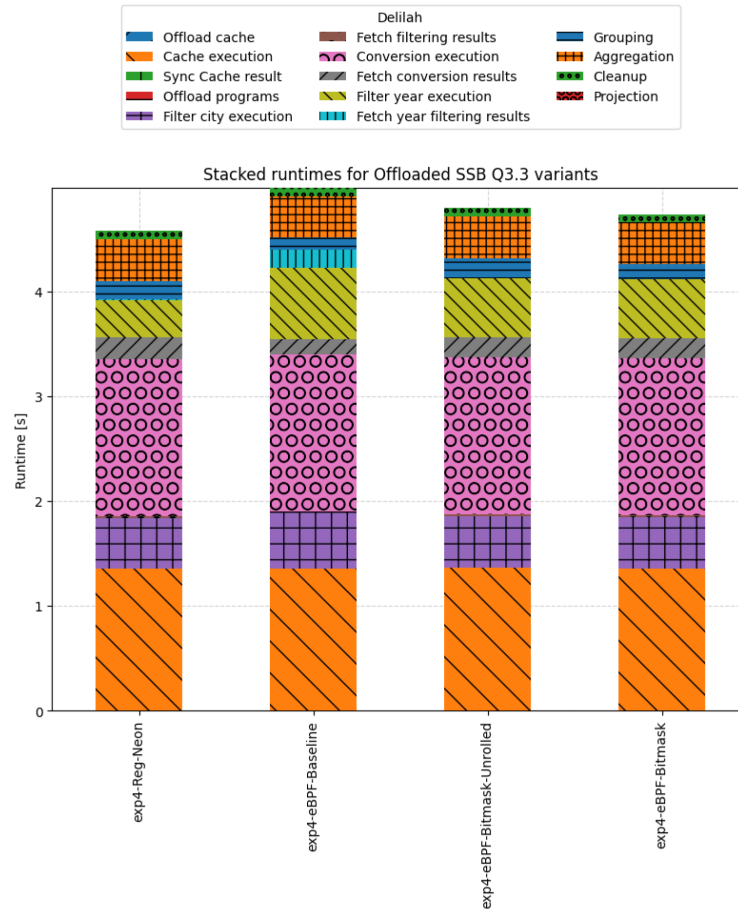


FIGURE 12.5: The result of running the full query on Delilah with various approaches to filtering.

efficiently?

12.5 Experiment 4: Filtering Strategies

In Figure 12.5, we explore four distinct implementations of the filter operator. The first implementation is a sequential eBPF baseline without specialised filtering optimisations. In the next experiment, we use bit masks instead of position lists, potentially enhancing data manipulation efficiency. On top of this, a third experiment integrates loop unrolling techniques into the bit-masked filtering process. Finally, we move on to bit masking with ARM Neon instructions, utilising the hardware capabilities available in the Xilinx MPSoC processor.

The first observable difference between the baseline implementation and the three implementations utilising bit masking is the performance improvement observed during data retrieval back to the host. Moreover, a notable reduction in the time required to filter over the input data is observable across all three bit-mask implementations.

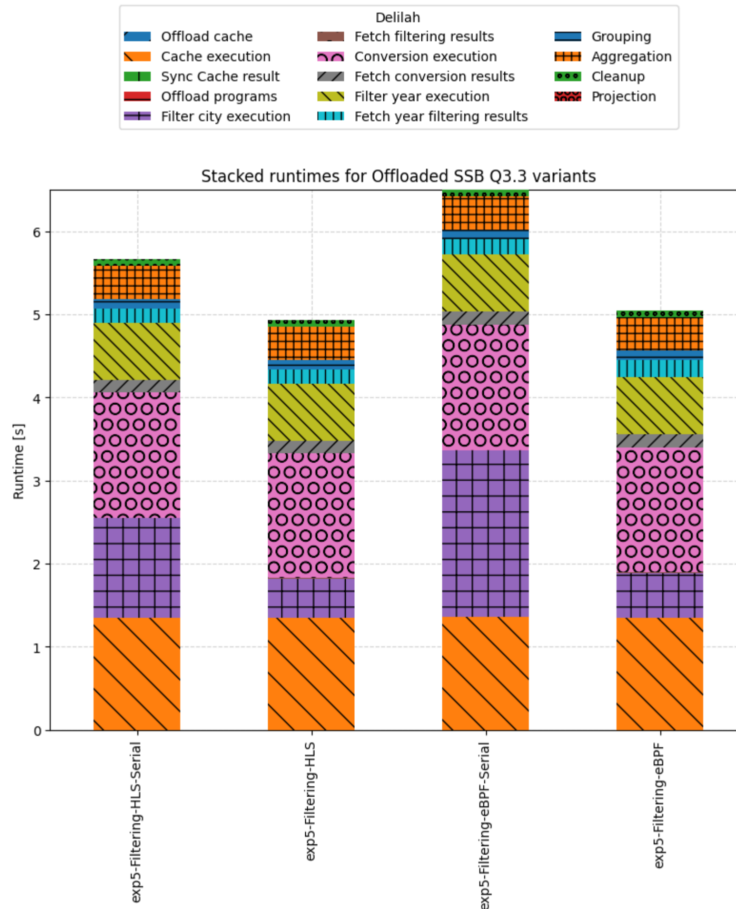


FIGURE 12.6: The result of running the full query on Delilah with hardware-accelerated filtering on the *Filter City* operation.

We understand that using bit masks plays a role in minimising the volume of data written into memory. Furthermore, when utilising bit mask techniques, the CPU caches will be filled up slower, thus improving bus utilisation.

We observe an exciting performance trend when enabling loop unrolling. Contrary to expectations, the unrolled version shows no improvement, with runtimes averaging 4.85 seconds, slower than the bit mask-only implementation, averaging 4.79 seconds. This unexpected result shows that typical compiler optimisations may have limited effect in an eBPF-based environment.

In conclusion, the design of data-intensive operators should prioritise maximising bus utilisation to optimise performance. Furthermore, it is essential to acknowledge that optimisations may not be automatically inferred by the compiler, requiring the developer to design operators that are efficient by design.

12.6 Experiment 5: Hardware Acceleration

In Figure 12.6, we focus on modifying the offloaded query to have a hardware-accelerated operation. This acceleration is achieved through High-Level Synthesis (HLS) and accessed using the Hardware module of Delilah, as described in Section 9.4.1.

Initially, our observations show a marginal difference in performance between hardware acceleration using HLS and the eBPF baseline. With both experiments concurrently executing four operators, HLS shows a slightly faster runtime of 5.72 seconds, compared to 5.85 seconds for the eBPF baseline.

To better understand the limited improvement in performance, we modify the experiment by introducing serial execution for the filtering process. In the modified version, the four operators are run serially, with only one active at any time. Under these conditions, we observe a hardware-accelerated runtime of 6.46 seconds versus 7.29 seconds for the eBPF baseline. This shift indicates a surprising aspect: while hardware acceleration does offer performance improvements, its impact is reduced when the underlying data path to memory becomes saturated.

In conclusion, hardware acceleration presents a path for enhancing performance. However, it is essential to weigh this against specific considerations. Hardware-accelerated HLS IPs are often specialised towards a single operation, and the investment of time and resources in implementing such IPs, coupled with increases in FPGA Look Up Tables (LUTs) and FPGA Flip-Flops (FFs), may not always justify the gains. Moreover, the trend indicating that the data path, rather than the execution itself, serves as the bottleneck suggests that efforts should be directed towards optimising data access latency rather than solely focusing on execution acceleration.

12.7 Experiment 6: Partial Offload

In Figure 12.7, we explore various degrees of offloading within the context of the SSB 3.3 query.

Firstly, the fully offloaded query implementation outperforms the implementation where aggregation is executed on the host. This outcome matches expectations, as device-side aggregation benefits from operating on data already available in the device's L1 and L2 caches, whereas the host-based approach requires fetching data before aggregation.

A notable observation shows on the host-only implementation, which spends considerable time on I/O operations from the underlying SSD. This time can be attributed to guaranteeing cache coherency using Selective Cache Invalidation (SCI), as described in Section 9.5. This characteristic is contrary to device-side I/O operations, which do not require visibility to the host, as data is kept on the device for the next operator.

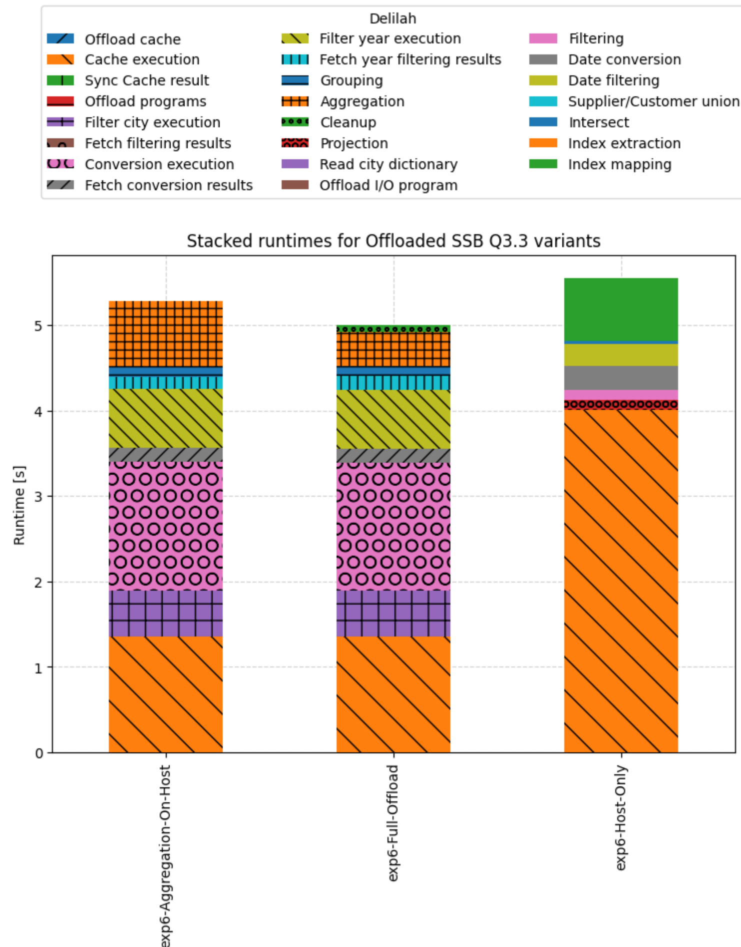


FIGURE 12.7: Three different degrees of offloading the SSB 3.3 query, spanning from full offload to partial offload with aggregation on host and host-only.

This trend prompts a thought-provoking question of the variability of computational storage. If cache coherency is attributed as the main reason for the success of computational storage devices, one may question said success. In the absence of cache coherency issues, as when using traditional NVMe SSDs, would the host-only implementation outperform the device, even when competing with computational storage optimisations such as bit masking and ARM Neon? The numbers of this experiment indicate that this would be the case.

In conclusion, deciding which operations to offload should be determined by the nature of the operators involved. When operations primarily involve data reduction, offloading proves advantageous. However, in cases where operations focus on transformation tasks over data reduction, conducting these transformations on the host may be more beneficial. Additionally, the notable impact of cache coherency on performance shows the importance of evaluating the feasibility of using computational storage.

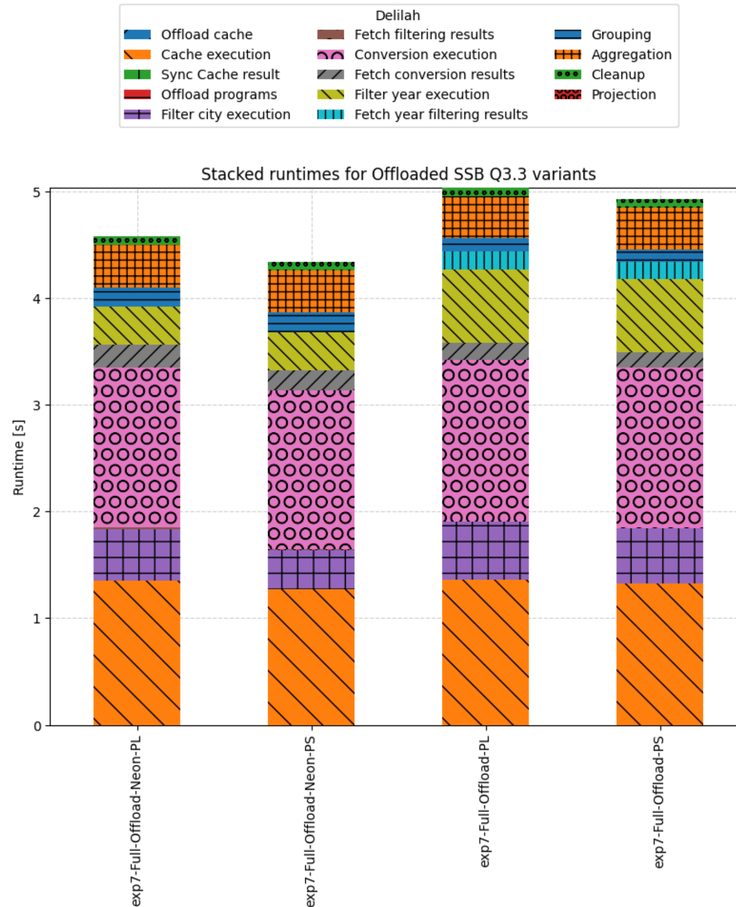


FIGURE 12.8: Two previous experiments, the baseline offloaded and ARM Neon implementations, with the shared data slot in PL and PS memory.

12.8 Experiment 7: Slot Placement

In Figure 12.8, we explore the placement of the shared data slot. Recall that the shared data slot is specialised and shared among all execution engines, often used to hold state or cache files that are reused in multiple executions. The host lacks access to the shared data slot, avoiding cache-coherency issues associated with the regular host-accessible data slots. In this experiment, we use the baseline eBPF implementation and the ARM Neon implementation.

Our observations in this experiment align with the findings of the memory latency experiments described in Section 11.2. Contrary to expectations, those experiments revealed higher latency on the Programmable Logic (PL) memory despite its significantly higher frequency than the Processing System (PS) memory. While the memory latency experiment was conducted as a microbenchmark experiment, this end-to-end experiment verifies the importance of memory latency in the success of computational storage devices.

To conclude, our findings reveal a lesson learnt: memory frequency should not be the determining factor in data slot placement. Instead, the focus

should be understanding the data path from the device's CPU to memory. Our experiment showed that the data path may be a more significant bottleneck than the memory frequency.

12.9 Discussion

The initial experiment demonstrates the performance differences between running Delilah in interpreted mode versus JIT mode. The order-of-magnitude difference in runtime for larger workloads emphasises the importance of JIT execution in optimising computational storage processor performance. The minor advantage of JIT in smaller workloads indicates that while interpreted mode may be sufficient for small programs, it is not practical for more complex programs. This trend supports the assumption that JIT should always be the default mode for computational storage processors to ensure optimal performance. From this, it also becomes evident that program execution should not be considered a single operation but a series of operations: one that transfers the program to the device, one that prepares the program (e.g. JIT compiling), and lastly, one that triggers the actual execution in the execution environment. Separating the execution command into several operations makes it possible to transfer and JIT-compile programs in advance, thus decreasing the preparatory work for the actual execution. It also makes it possible to re-execute the same JIT-compiled program multiple times, which is unsupported in Delilah since the preparation and execution happen within the same command.

The experiments on partitioned execution show the non-triviality of dividing workloads. While partitioning intends to utilise host processing capabilities and the computational storage device, it causes significant overheads in communication, cache management, and shared memory allocation. These overheads often negate the benefits of partitioning, making it more beneficial for computational storage processors to complete as much work as possible without artificially partitioning the workload. This observation shows the need to fully understand the consequences of workload partitioning to avoid unnecessary overhead.

The experiment on filesystem caching shows the benefits of using direct I/O operations in computational storage processors. Direct I/O bypasses the filesystem cache, improving the performance of read operations. However, while direct I/Os may improve performance, they add complexity since the underlying operation must be aligned and of a specific size, and the memory must support such operations. In Delilah, only the PS memory supports this operation.

While exploring different filtering strategies, we learnt that using bit masks and ARM Neon instructions significantly improves performance. This increase in performance is likely due to more efficient memory access patterns. However, the lack of expected gains from loop unrolling indicates that typical compiler optimisations are ineffective in an eBPF-based environment.

This experiment shows the importance of manually designing efficient operators and not relying on default compiler optimisations.

The hardware acceleration experiment reveals that while High-Level Synthesis (HLS) can offer performance improvements, the benefits may be limited by data path saturation. While executing in parallel, the minor performance difference between hardware acceleration and eBPF indicates that optimising data access latency is as essential as improving execution time. Furthermore, this shows the need to balance the investment in hardware-specific optimisations and the potential gains.

The experiment on partial offloading of the SSB 3.3 query shows similar results to the experiment on partitioning. Aggregation decreases the data transfer between the computational storage device and the host. In turn, this decrease also decreases the need for cache coherency guarantees. Reducing DMA operations is critical in optimising cache coherency performance. This discovery suggests that computational storage devices should initially focus on reducing data movement between the device and the host.

Finally, the slot placement experiment shows the direct consequence of memory latency, which we discussed in Chapter 11. Despite the higher frequency of Programmable Logic (PL) memory, the higher latency than Processing System (PS) memory suggests that the data path, rather than memory frequency, is the primary bottleneck. This finding emphasises the need to understand the entire data path from the device CPU to memory when designing computational storage processors, rather than focusing solely on memory specifications. On the Daisy OpenSSD and in Delilah, this means that performance is generally improved by looking at clock frequencies and congestion on the FPGA, rather than by inserting higher-frequency memory modules or increasing processor frequency.

In essence, we can summarise the experiments into a set of design principles.

Design Principle	Experiment	Assumption
Prefer Just-in-Time over interpretation.	Experiment 1: Both Just-in-Time (JIT) compilation and interpretation have the same baseline performance, but JIT scales better.	Interpreting a single eBPF instruction costs multiple native instructions.
Maximise operator workload.	Experiment 2: Operators should process as much data as possible in a single pass.	The overhead of offloading and scheduling eBPF programs is high due to the time spent transferring and preparing the execution engine.
Avoid operating system overhead.	Experiment 3: I/Os perform better if they avoid OS overhead, e.g., the filesystem cache.	Memory latency is high, and removing overhead optimises performance.
Reduce memory accesses.	Experiment 4: Memory-intensive operations underperform.	Memory latency is high, and limiting memory accesses optimises performance.
Hardware-accelerate where possible.	Experiment 5: Hardware acceleration can avoid bottlenecks in data paths.	Memory latency is high, and hardware acceleration helps limit memory accesses and improve performance.
All or nothing offload.	Experiment 6: Offloading only parts of a query leads to increased overhead.	The overhead of offloading and scheduling eBPF programs is high due to the time spent transferring and preparing the execution engine.
Prioritise data path capacity.	Experiment 7: The data path from CPU to memory tends to congest before the memory module itself.	Memory is connected via AXI interface and PS-to-PL ports.

TABLE 12.1: Design principles and assumptions derived from experimenting with Delilah on the Daisy OpenSSD.

Chapter 13

Lessons Learnt

This chapter describes the lessons learnt from designing, implementing, and evaluating Delilah.

13.1 Memory Management

Memory management is a crucial open question when combining the functionality of computational storage and integrated data analysis pipelines. With the emergence of multi-core processors, it is no secret that adding extra processing capabilities introduces the need for cache-coherency mechanisms. However, with computational storage, the challenges are more significant due to the isolation of components.

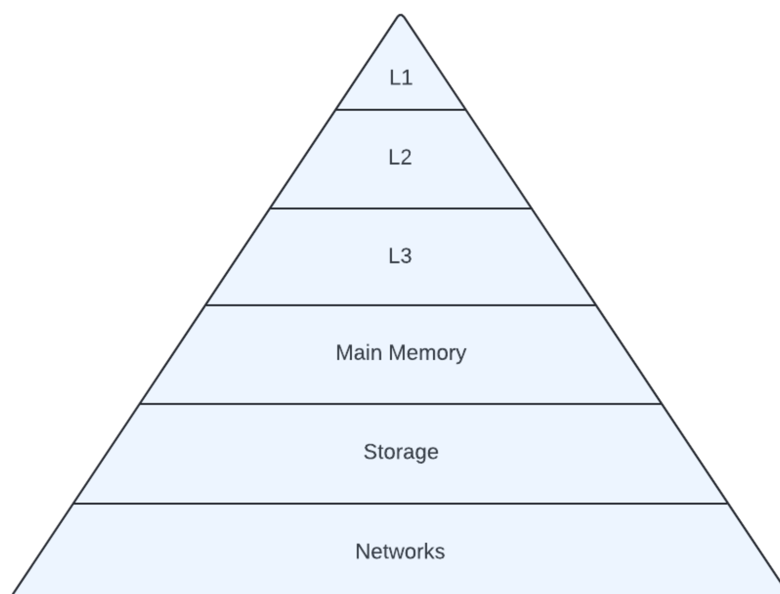


FIGURE 13.1: The traditional understanding of memory hierarchy, where the upper levels are always subsets of the levels below.

Figure 13.1 shows the traditional understanding of the memory hierarchy of multi-core processors. Here, the memory of higher levels in the memory

hierarchy are always a subset of lower levels. The typical way to handle invalidation for multi-core processors is to mark the memory areas that are now outdated explicitly. Subsequently, when the processor core accesses this memory, it sees the mark and fetches it from the lower levels.

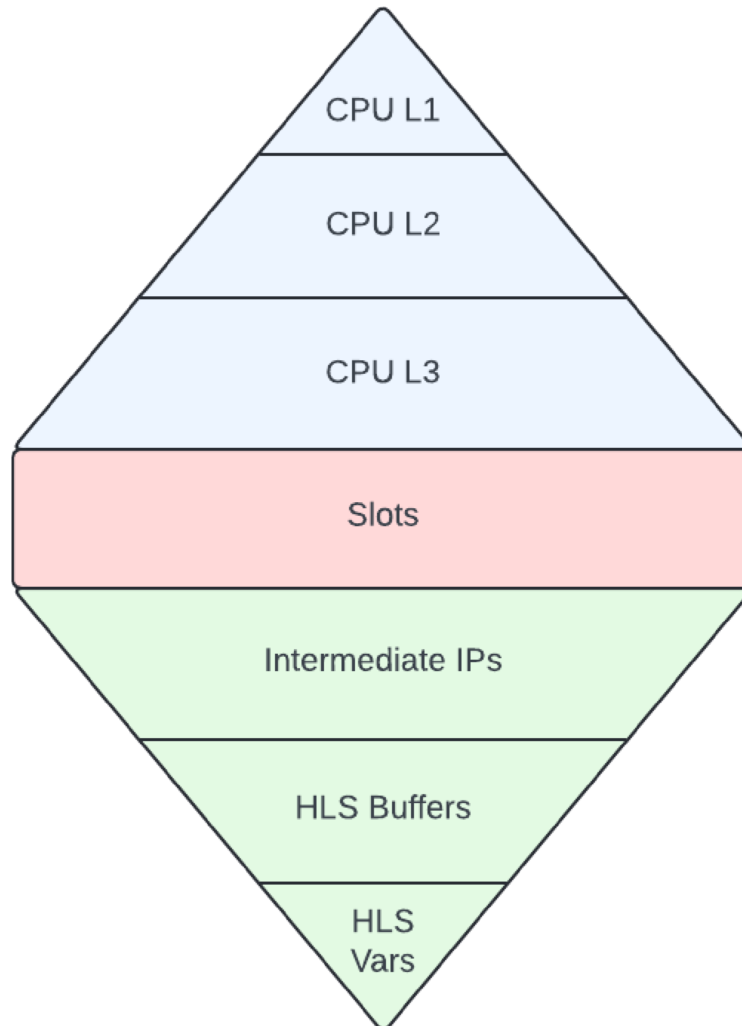


FIGURE 13.2: The new understanding of memory hierarchy with computational storage, where multiple memory hierarchies share a lower layer but have separate upper layers. This complicates moving data in and out of various levels of caches across a collection of compute units.

However, computational storage is more complex. Organising the memory into a hierarchy becomes non-trivial, even on the device itself. Figure 13.2 shows an example of a computational storage device with a processing unit and an accelerator IP. The processor and the accelerator may share the same underlying data slots. However, the CPU's local caches are unaware of intermediate IPs (like AXI connectors or buffer IPs) and local buffers and variables in the accelerator. It is not clear how to understand and organise the

hierarchy efficiently.

It becomes even more unclear when including the host in the understanding of the memory hierarchy. For example, are the data slots of the computational storage device a subset of storage? With this understanding, the host memory must then be a subset of computational storage, which is not the case since there is not necessarily an overlap between host memory and computational storage memory.

While understanding the memory hierarchy is not inherently a problem, it leads to several, critical underlying questions. One issue is how memory hierarchies, which do not share upper levels, manage data movement in or out of various levels of caches across a collection of compute units on both the host and the device. This separation introduces the critical problem of synchronising data between different cache levels.

In traditional memory hierarchies, invalidating a cache line at the upper level would automatically trigger the data to be fetched from lower levels again when needed. However, the memory hierarchy of computational storage implies that lower levels might have the same outdated data, and synchronisation with the lowest level becomes necessary to ensure consistency. The challenge is compounded by the fact that the lowest level of the memory hierarchy might be located at a significant distance from the upper levels. This distance could be physical, involving traversal across PCIe links or other interconnects, which introduces additional latency.

In summary, while the fundamental concept of memory hierarchy is straightforward, the practicalities of managing data movement and synchronisation across non-shared upper levels introduce several critical challenges. These challenges revolve around ensuring data coherence. Organising and effectively building memory hierarchies for computational storage is a critical challenge that must be addressed. Until it is addressed, computational storage devices will face allocation and cache coherency issues.

13.1.1 Delilah and NVMe

After experiencing memory management challenges with Delilah, we better understand the limitations of the NVMe specification. The NVMe specification introduces two mechanisms but leaves critical implementation details up to the vendor.

First, it needs to be clarified how synchronisation and cache coherency work with the introduction of Memory Range Sets and, thus, overlapping memory regions. What happens if two programs operate on the same memory from two different namespaces? What if the host initiates a transfer while a program operates on a memory region? How is coherency guaranteed between namespaces?

Secondly, with the introduction of RTL, eBPF, and other mechanisms for program offload, the specification raises more questions than it answers. Even a

simple eBPF execution environment (e.g., uBPF) has several configurations, like execution strategy (JIT vs. interpreted) and registered functions that must be managed. With FPGAs, the number of parameters are much higher. How does the host manage the execution environment? How are the limitations or restrictions of the execution environment propagated?[1]

Furthermore, there is also the question of coherency between NVMe itself and the execution environments. This question, in turn, raises even more questions: What happens if two different types of computational programs access the same memory? How is cache coherency guaranteed between these potentially very diverse execution environments?

13.2 eBPF Complements Computational Storage

Our experiences with eBPF in the context of computational storage have shown us that eBPF can organise and express many different operators with varying characteristics. For example, designing and implementing almost any bounded single-pass algorithm is trivial. We have, however, learnt that eBPF cannot stand on its own. There must be a mechanism to intertwine the vendor-neutrality and lightweight characteristics of eBPF with device-specific functionality. The usage of the `call imm` instruction in eBPF has proven to be a viable mechanism to switch between these two domains of programs.

However, open questions concerning offloading eBPF to computational storage still need to be answered. For example, in the Delilah architecture, we use function names to map device-specific functions. In our architecture, when compiling to eBPF, the resulting ELF will contain a mapping table from the function number in the `call imm` instruction to the function name, which the execution environment can invoke. This mechanism means that developers of computational storage programs must know, at compile time, which functions are registered on the targeted computational storage device. Subsequently, this means that the resulting eBPF ELF binary is not vendor-neutral but instead compiled for a specific device. This challenge also raises several questions on exchanging device capabilities, even before the target device is connected or known. How does the developer know what functions are present on the device? What happens if a device typically has a given registered function but is currently unavailable for external reasons (e.g., incompatible underlying storage or resource constraints)? What happens if an unknown function is offloaded to the device?

Furthermore, we have learnt that error handling in eBPF is another critical question to solve. When using eBPF in the kernel context, we often have

[1] eBPF programs and execution environments may have many subtle limitations or differences. For example, eBPF programs have a 512-byte stack by default, but since this may be limiting for complex programs offloaded to complex environments like storage, it can now be overridden in LLVM [22]. How do the execution environment and the host manage the eBPF stack size? What happens if they are configured with different stack sizes?

simple and restricted programs with limited outcomes. Here, it is typically sufficient to communicate with the host program through return values or eBPF maps. However, in computational storage, the number of outcomes is larger and more complex to express. Assume that eBPF invokes a device-specific program or registered function to read a file from the underlying storage. Now assume this read fails for any reason (e.g., file does not exist, alignment issues, permission denied). A registered function only has a scalar return value; how does it inform the eBPF program that the I/O has failed, and what, in particular, was the underlying problem? A partial solution could be to provide a data structure as a parameter to the function that can contain errors if applicable. However, we merely moved the problem, as we are now left with questions on what attributes such data structure should contain, and where it should be allocated. Furthermore, eBPF does not have a mechanism to return anything but a scalar value to the host, so it is up to the developer to return the device error information to the host via data slots.

Lastly, while eBPF is a promising instruction set architecture for computational storage, we are left with some critical limitations. For example, the lack of floating-point operations severely limits the operators that can be offloaded. There is a natural tension with eBPF being restricted for safety purposes in the kernel and the need for expressibility with program offload. This tension is unlikely to be resolved due to the opposing forces of the use cases.

We can summarise the open questions of eBPF in the context of computational storage into a few bullet points:

- How do the host and computational storage devices exchange device capabilities? How are the capabilities known at compile time?
- How can registered functions yield error information to the eBPF program? How can the eBPF program propagate the error information to the host?
- How do we manage the natural tension between eBPF as an extension of the kernel and eBPF as the preferred instruction set architecture for computational storage?

13.3 Workload-Awareness Is Key

While eBPF complements computational storage despite its limitations, we have also learnt that not all workloads can be efficiently offloaded to the storage layer. While this may seem obvious, the critical point is that even different implementations of the same operators can have widely different performance characteristics.

For example, we have learnt that interpreting eBPF is inefficient and that eBPF code should always be JIT-compiled to native processor instructions. We have learnt that the overhead of offloading eBPF programs and preparing the execution environment puts pressure on the developer to coalesce as many operators as possible and only partition where necessary.

We have learnt that computational storage designers should spend time optimising memory throughout and latency, rather than execution throughput and latency. Our experiment with HLS and ARM Neon showed that performance bottlenecks are typically related to memory and not processing frequency.

At the same time, throughput is also a key consideration when designing operators. While latency limits how often we can execute a given operator, throughput determines how much data we can process in a single execution. Improving throughput involves several fundamental design considerations. For instance, avoiding multiple passes on a dataset is essential, as each additional pass effectively halves the data sizes processed in a single execution. Single-pass algorithms reduce the time complexity and minimise the memory bandwidth consumption, leading to higher operator throughput.

As such, when determining which workloads are suitable for offload to the storage layer, one must consider the following characteristics:

- **Locality**: If the workload has low locality, it is better to execute the operations host-side. This is because the host has a higher frequency and lower memory access latencies and thus handles cache misses and unpredictable access patterns better.
- **Data Reduction**: If the workload does not significantly decrease the data transferred to the host, it is better to execute the operations host-side. The lower frequency of the device processor and higher memory access latencies are partly made worthwhile by the decrease in DMA transfer time. If data volume is not decreased, the offload may not be worthwhile.
- **Single Pass**: The throughput will decrease if the algorithm requires multiple passes. Requiring too many passes may decrease the throughput to a degree where executing the operations on the host is better. This is the case because the host often has lower memory latency and larger CPU caches, and thus performs better when passing over significant amounts of data multiple times.

Of course, the optimal offloading strategy depends on the cooperation between the device and the host. For example, it is worth offloading inefficient operations to free up host processing capacity if this reduces the total time of all operations.

Concerning integrated data analysis pipelines, the lessons learnt indicate that automatically generated operators are too risky to offload. If the generated operator does not access data sequentially and predictably, the operator will not be worth offloading compared to executing directly on the host. As such, to integrate with eBPF-based computational storage devices, integrated data analysis pipelines should have a set of operators known to be efficient on the target device. These operators could be filtering or aggregation, which are single-pass and have a predictable access pattern. At the same time, this also

minimises the work spent building a scheduler that can generate eBPF code on the fly.

Furthermore, if automatically generated eBPF code is the preferred approach for a given pipeline, time should be spent verifying the generated code's performance, and guaranteeing that the generated code does not perform undefined or risky operations.

Chapter 14

Summary

In Part III of this thesis, we have experimented with Delilah. In Chapter 11, we explored the memory characteristics of the Daisy OpenSSD with micro-benchmarks, and in Chapter 12, we offloaded SSB 3.3 to Delilah with various configurations. Lastly, in Chapter 13, we described the lessons we have learnt from experimenting with Delilah.

The experiments revealed memory access characteristics for computational storage devices like Delilah when deployed to the Daisy OpenSSD. We analysed Direct Memory Access (DMA) versus Controller Memory Buffers (CMB) to demonstrate that while CMB is efficient for small data transfers, DMA performs better for larger transfers, specifically when multiple channels are utilised. Access patterns play an essential role, with sequential accesses benefiting from cache prefetching and exhibiting minimal latency, whereas random accesses suffer from increased latency due to frequent cache misses. The inefficiencies observed in memory accesses align with existing research, stressing the importance of understanding and optimising memory access patterns before selecting the operators to offload to the storage layer.

The exploration of Delilah's performance in interpreted mode versus Just-In-Time (JIT) compilation mode shows the significant advantages of JIT for larger workloads, suggesting that JIT should be the default mode for computational storage processors (CSPs). The partitioned execution experiments reveal the complexities and overheads associated with dividing workloads, indicating that computational storage processors should aim to minimise unnecessary partitioning to avoid communication and cache management overheads. Direct I/O operations improve read performance by bypassing filesystem caching, and hardware acceleration offers limited benefits due to data path saturation.

The chapter on lessons learnt highlights the complexities of memory management in computational storage systems, particularly the challenges of maintaining cache coherency and organising memory hierarchies. The chapter identifies critical limitations of the NVMe specification, such as synchronisation issues with overlapping memory regions and the complexities introduced by different execution environments. Using eBPF for computational

storage shows promise but entails several open questions, including the need for efficient error handling and managing device-specific function mappings.

The chapter demonstrates the importance of workload awareness in optimising computational storage. Efficient offloading strategies depend on understanding the locality and data reduction characteristics of workloads. Furthermore, the need to fully understand the offloaded workload implies that automatically generated eBPF code is not a viable approach. Instead, developers of integrated data analysis pipelines should design and implement hand-tailored operators that are efficient on the target hardware.

The characteristics of Delilah and eBPF, and how they all play together with an integrated data analysis pipeline, is too early to conclude upon. We now know that programs can be efficiently offloaded to the storage layer via eBPF, and that long-term memory organisation between operators is possible. However, we have yet to understand how to integrate with pipelines efficiently. Many open questions have yet to be answered. How can an integrated data pipeline schedule offload to storage efficiently? How does the pipeline organise data and state efficiently? Can operators be generated automatically in an efficient way? All these questions are considered open for future work.

Chapter 15

Conclusion

This thesis aims to connect two merging technologies: integrated data analysis pipelines that connect frameworks from multiple domains, and eBPF, a vendor-neutral lightweight instruction set architecture (ISA).

In the first part, we thoroughly surveyed the state of the art to understand how contemporary computational storage devices look, and how they may support workloads from different host applications. We saw that computational storage has yet to gain significant traction due to the lack of standardised interfaces and implementation limitations like short-lived, non-stateful memory. Furthermore, we explored how host applications are currently built and which characteristics seem essential for consumers of large workloads. Lastly, we identified the Daisy OpenSSD as a testbed for computational storage.

In the second part, we implemented Delilah, the world's first public implementation of an eBPF-based computational storage device. Building Delilah led us to understand better the challenges, opportunities, and limitations of computational storage. With Delilah, we encountered several open questions concerning memory management and cache coherency. We challenged the current understanding of memory hierarchies to include multiple hierarchies growing from the same foundation, but in different directions. This new understanding emphasises the complexities of memory management, especially concerning cache coherence. While we proposed SCI as a potential solution, more must be done to resolve the underlying problem. How do we efficiently manage cache coherency with multiple overlapping hierarchies growing in different directions?

By resolving these issues, it becomes possible to integrate computational storage devices like Delilah efficiently into integrated data analysis pipelines, which opens up a wide range of new optimisation avenues. For example, since memory lives between offloaded programs, it will be possible for the integrated data analysis pipeline to make state and intermediate data available to subsequent executions. Furthermore, since the pipeline now has a more exhaustive view of the memory layout of computational storage, it will be possible to use it to start upcoming operations ahead of time, a form of prefetched I/O operation. With this architecture, the computational storage

device could read and transform data to immediately be ready for more complex processing on the host.

In the third part of the thesis, we explored Delilah's initial performance characteristics and experimented with offloading a database query. Here, it became clear that the memory management challenges discovered from the implementation broadly impact performance. Delilah suffers from significant memory latencies when deployed to the Daisy OpenSSD, and these latencies, in turn, make operations like memory management very costly – too costly for some operations to be worth offloading.

However, despite our challenges, we saw that Delilah could significantly improve performance when handling some operations like filtering. We saw that when memory optimisations like bit-masking are applied, Delilah performs better than the host. How Delilah or other computational storage devices perform alongside an integrated data analysis pipeline is still an open question.

Glossary

API Application Programming Interface. An API is a collection of functions developers can use to interact with a framework, library, device, or other system. APIs are often stable and do not change. The underlying implementations, however, may change to improve stability or performance. 11, 15, 17, 35, 42, 43, 45, 59

ARM Advanced RISC Machine. A specific implementation of the RISC architecture, which has gained significant traction in recent years. x, xi, 2, 8, 14, 15, 26, 28, 30, 38, 47, 48, 57, 76, 93, 96–98, 101, 104–106, 114

ASIC Application-Specific Integrated Circuit. Like FPGAs, ASICs offer a set of programmable logic blocks. However, in contrast to FPGAs, ASICs do not allow reprogramming of the device, thereby increasing security and reliability. 23

AXI Advanced eXtensible Interface. An on-chip communication bus protocol developed by ARM. It is often used to connect IPs in FPGAs. 31–33, 36, 37, 78–81, 90, 93, 94, 108, 110

BAR Base Address Register. BARs provide hardware registers for establishing a shared memory-mapped region between host and device. 18, 29, 33, 56, 63, 68, 70, 88, 90

BPF Berkeley Packet Filter. A simple ISA originally designed for expressing network packet filters. 2, 8, 10, 12, 14, 47

BRAM Block Random-access Memory. BRAM is a type of IP that exposes the internal logic gates of an FPGA as system memory. 56, 81

CMB Controller Memory Buffer. CMBs are a mechanism in NVMe to access PCIe BARs. The main goals of CMBs are placing queues in host memory and placing data for DMA in host memory. xii, 88, 94, 116

Computational Storage The concept of moving data processing closer to where the data physically resides. Computational storage aims to free up host-side processing capacity and to reduce data movement across system bottlenecks. ii, iii, x, xi, 1–5, 11, 13–16, 18–22, 25, 26, 28, 44, 45, 47–50, 53, 55, 61, 85, 86, 97, 104, 109–114, 116–118

Computational Storage Array (CSA) A collection of computational storage devices, control software, and optional storage devices. A computational storage array provides computational capabilities to potentially diverse devices. 20, 22

Computational Storage Device (CSx) A computational storage drive, computational storage processor, or computational storage array. ii, 2, 4, 5, 13–16, 20–23, 26, 47–49, 52, 53, 55, 57–60, 85, 86, 100, 104–107, 110–114, 116, 118, 119

Computational Storage Drive (CSD) A storage element that provides computational storage functions and persistent data storage. The main difference between computational storage arrays and computational storage drives is the number of underlying persistent storage mediums. 20–22, 49, 58

Computational Storage Engine (CSE) A component that can execute one or more computational storage functions. A computational storage engine is a collection of execution environments and device-specific functions compatible with these environments. 20, 21, 23

Computational Storage Engine Environment (CSEE) An operating environment for a computational storage engine. A computational storage engine environment could be a virtual machine, accelerator or any other unit with computational capabilities. 21, 23

Computational Storage Function (CSF) Specific operations that may be configured and executed by a computational storage engine. A computational storage function is a concrete device-specific or host-offloaded procedure compatible with at least one of the execution environments embedded within the computational storage device.. 20–23

Computational Storage Processor (CSP) A device that provides computational storage functions for an associated storage system without providing persistent data storage. The main difference between computational storage processors and computational storage drives is the fact that computational storage processors do not have persistent storage. Instead, they are connected to persistent storage via PCIe, Ethernet, IB, or similar interfaces. ii, 1, 2, 5, 21, 22, 49, 53, 55–58, 80, 85, 96–100, 106, 107, 116

Computational Storage Resource (CSR) A resource available for a host to provision on a computational storage device that enables that computational storage devices to be programmed to perform a computational storage function. Computational storage resource is a catch-all term for any subsystem on a device that supports computational storage capabilities. 21, 23

CQE Completion Queue Entry. A CQE results from an asynchronous I/O popped from a completion queue of io_uring. When io_uring has consumed and processed an SQE, it will emit a CQE. 59

DAG Directed Acyclic Graph. A type of graph with directed edges between vertices which will never form a closed loop. 42, 43, 48

Daisy OpenSSD The Daisy OpenSSD is the fourth generation of the OpenSSD project, built to promote research and education on recent SSD technologies [55]. The Daisy has several peripheral connectors and a Xilinx MPSoC. x, xii, 2–4, 28–32, 50, 56, 58, 60, 62–64, 78, 86, 90, 91, 93, 94, 97, 98, 107, 108, 116, 118, 119

DAPHNE DAPHNE is a Horizon 2020 project which aims to build a system infrastructure for integrated data analysis pipelines, including data management and processing, high-performance computing (HPC), and machine learning (ML) training and scoring [16]. x, 4, 43–46, 48, 49, 55, 61

DDR Double Data Rate. A computer bus using DDR transfers data on the rising and falling edges of the clock signal simultaneously. It is often used in the context of computer memory. 28, 39, 49, 56, 60, 64, 79, 84, 90, 93, 94, 96

DIMM Dual In-line Memory Module. DIMM memory is a standardised type of computer memory. In everyday talk, they are often referred to as memory sticks. 33, 56, 91

DMA Direct Memory Access. DMA is a system for transferring large amounts of data to peripherals while bypassing the central processing unit. x, xii, 18, 29, 31–33, 37, 38, 56, 60, 63, 66, 83, 88–90, 94, 107, 114, 116

eBPF Extended Berkeley Packet Filter. A redesign of the original BPF aimed at contemporary hardware for improved performance. eBPF is furthermore not limited to network packet filtering but can also be used for performance monitoring and security auditing. ii, iii, xii, 1–5, 8–14, 18, 19, 22, 23, 25, 26, 47–50, 52–58, 60, 61, 66, 69, 70, 72, 74, 75, 83, 85, 86, 95–97, 101–103, 105–108, 111–118

Eid-Hermes Eid-Hermes is an open-source eBPF accelerator developed by Eideticom. The aim of Eid-Hermes is to explore how eBPF-based accelerators can execute code from a host processor [40]. xii, 2–4, 14, 18, 19, 26, 47, 58–60, 64–68, 70, 83, 85, 86, 88

ELF Executable and Linkable Format. ELF is a flexible, extensible and cross-platform format designed for executable files, object code, shared libraries and core dumps. 10, 47, 70, 112

FPGA Field-programmable Gate Array. FPGAs have a set of programmable logic blocks and a hierarchy of reconfigurable interconnects that allow

blocks to be wired together. Many FPGAs can be reprogrammed to implement different logic functions, allowing features like those within traditional computer software. 1, 14–20, 22, 23, 25–28, 30–36, 38, 39, 47, 48, 55, 57, 61, 64, 79, 80, 82, 84, 86, 96, 103, 107, 112

Function Data Memory (FDM) Device memory used for storing data that is used by the computational storage functions and is composed of allocated and unallocated function data memory. 22, 23

GCC GNU Compiler Collection. GCC is a collection of compilers built for various programming languages, hardware platforms and operating systems. It is one of the major compilation toolchains, with LLVM being the other. 8, 34

GPIO General-purpose Input/Output. A simple system for transmitting signals on pins or wires. 31, 33, 34, 71, 78, 79, 83, 84

GPL GNU General Public License. GPL is an open-source license. Contrary to BSD or Apache licenses, GPL require users of GPL code to disclose their code, thus limiting the applications in for-profit use cases. 8, 10, 47

hBPF Hardware Berkeley Packet Filter. A user-space VM for parsing, JIT-compiling and interpreting eBPF code, targeted at FPGAs. 10, 11, 57, 58

HDF Hardware Design Files. A file describing the hardware components of an FPGA. HDFs are often created from block designs in programs like Vivado. Contrary to HDL, HDFs describe the programmable part of the FPGA. 32, 34

HDL Hardware Description Language. HDL is a domain-specific language for expressing the hardware components of an FPGA, i.e. the capabilities and constraints of an FPGA. 35, 36

HLS High-Level Synthesis. HLS allows developers to write C programs and compile them into FPGA IPs. The resulting IPs can be utilised alongside other IPs in a block design. 31, 36, 37, 53, 70, 71, 79–82, 103, 107, 114

io_uring io_uring is a Linux system call interface for asynchronous I/Os, built to address performance issues with synchronous interfaces like `read()` and `write()`. io_uring makes use of both submission queues (see SQE) and completion queues (see CQE). 4, 59, 60, 66, 83

IP Intellectual Property. An IP core is a block of logic wired to other IPs and deployed on an FPGA. IPs can do simple logical operations or manage complex tasks like integrating with FPGA peripherals. 4, 31–34, 36, 37, 65, 67, 70, 71, 78–80, 82, 84, 86, 103, 110

IRQ Interrupt ReQuest. An IRQ is a hardware signal sent to the processor that temporarily stops a running program while executing another

program. It is often used to signal that some event has occurred on a peripheral device. 71

ISA Instruction Set Architecture. An abstract model of a computer architecture that defines the supported instructions, data types, registers, fundamental features and input/output model. 2, 9, 52, 53, 85, 118

JIT Just-in-Time. Just-in-Time compilation is the concept of compiling programs dynamically at runtime. JIT is often done if the underlying architecture of the execution environment is unknown at compile-time. x, 2, 8, 11, 25, 42, 47, 57, 67, 96–98, 106, 108, 112, 113, 116

LLVM Low-Level Virtual Machine. LLVM is a collection of compiler and toolchain technologies that can be used to develop frontends for any programming language and backends for any ISAs. It is one of the major compilation toolchains, with GCC being the other. 8, 112

MIG Memory Interface Generator. A specific IP from Xilinx for exposing a DDR memory stick as an AXI backend. 33, 78–80, 84, 90, 93

MPSoC Multiprocessor System on a Chip. MPSoCs are a specific processing unit type with both a PS and a PL domain. These provide the benefits of FPGAs and traditional computers in the same device. 20, 28, 30–34, 36, 38, 39, 48, 61, 68, 80, 83, 93, 101

NVMe Non-Volatile Memory Express. NVMe is a protocol for managing non-volatile storage devices attached via PCIe. ii, iii, 13, 16, 20, 22–24, 26, 28, 34, 38, 39, 48, 49, 58, 60, 78, 86, 104, 111, 112, 116

PCIe Peripheral Component Interconnect Express. PCIe is a serial computer expansion bus often used for graphics cards, hard disk drive host adapters, SSDs, Wi-Fi and Ethernet hardware connections. 1, 3, 4, 15–18, 21, 28, 29, 32, 33, 38, 48, 49, 53, 55, 58, 60, 65, 68, 77, 78, 90, 96, 111

Petalinux The Petalinux SDK, created by Xilinx, enables developers to package a light-weight operating system and deploy it to a CPU embedded within Xilinx MPSoCs. 1, 4, 31, 33–35, 61, 63, 68, 79, 83, 84

PL Programmable Logic. The PL domain of an MPSoC contains the FPGA component and the IPs. xi, xii, 30, 32, 33, 56, 60, 61, 74, 76, 91, 93, 94, 96, 100, 105, 107, 108

PS Processing System. The PS domain of an MPSoC contains an operating system and a CPU, and it interfaces to PL. xi, xii, 30, 32, 33, 56, 60, 61, 63, 74, 76, 78, 79, 90–94, 100, 105, 106, 108

QEMU Quick Emulator. A Linux emulation tool for emulating hardware. It is often used while developing drivers and modules when the target hardware is unavailable to test on. 18

rBPF Rust Berkeley Packet Filter. A user-space VM for parsing, JIT-compiling and interpreting eBPF code, written in Rust. 11, 57

RISC Reduced Instruction Set Computer. A computer architecture built for low cost, minimal power consumption and low heat generation. It archives this by reducing the number of distinct operations the processor can execute. 2, 10, 47

SDK Software Development Kit. A set of tools and packages for writing software to a particular device or platform. 1, 31, 32, 34, 84

SNIA Storage Networking Industry Association. SNIA is a registered non-profit that develops global standards and delivers education on all technologies related to data. ii, iii, x, 1, 20–23, 26, 48, 49, 86

SQE Submission Queue Entry. An SQE is an asynchronous I/O inserted into a submission queue of io_uring. When io_uring has consumed and processed the entry, it will emit a CQE. 59, 67

SSB Star Schema Benchmark. The Star Schema Benchmarks is a set of SQL queries built to measure the performance of database systems. The performance of the queries can be used to compare various characteristics of diverse database systems. xi, 5, 6, 94–96, 103, 104, 107, 116

SSD Solid-State Drive. A type of storage device with predictable microsecond latency and high throughput. SSDs have no moving parts, contrary to traditional hard disk drives. xii, 1, 13–16, 18, 28, 31, 32, 34, 38, 48, 55, 77–79, 96, 103, 104

TSL Template SIMD Library. TSL is a C++ template header-only library built for abstracting away device-specific instruction sets with a particular focus on SIMD (Single Instruction, Multiple Data). In essence, TSL enables developers to express optimised instructions abstractly and compile them down to device-specific instructions. 53, 62, 76, 83

uBPF User-space Berkeley Packet Filter. A user-space VM for parsing, JIT-compiling and interpreting eBPF code, written in C. 8, 10, 11, 18, 47, 57, 58, 60, 62, 67, 70, 72, 83, 96, 97, 112

UDMA UDMA or *u-dma-buf* is a kernel module that easily maps kernel memory to user-space and manages cache behaviour and coherence. In Delilah, we use UDMA to expose reserved memory as slots from the kernel to Delilah. x, 37–39, 63, 74, 75, 84, 91, 92, 100

Vivado Vivado, which comes in a standard and HLS edition, is used to program FPGAs. In the standard edition, developers can organise hardware components, called IPs, as a diagram of blocks. In the HLS edition, developers can write C programs to be compiled into FPGA hardware components. 31, 32, 35–37, 70

VM Virtual Machine. A virtual machine emulates a computer architecture and the related hardware only using software. 2, 8, 10, 21, 47, 57, 72, 83

x86 x86 and x86_64 are the most common ISAs. The first iteration of x86, x86_16, was initially developed by Intel in 1978. 2, 8, 47, 57

XDMA Xilinx Direct Memory Access. XDMA is a specific implementation of DMA for Xilinx FPGAs. 4, 18, 29, 32, 33, 65–68, 71, 78–80, 84, 90

Bibliography

[1] Martın Abadi et al. "TensorFlow: A system for large-scale machine learning". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. Savannah, GA, USA, Nov. 2016. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf (visited on 05/17/2024).

[2] *An introduction to AMBA AXI*. URL: https://developer.arm.com/documentation/102202/0200/AXI-protocol-overview (visited on 05/11/2022).

[3] *AXI Bridge for PCI Express Gen3 Subsystem v3.0 Product Guide*. July 2020. URL: https://docs.xilinx.com/v/u/en-US/pg194-axi-bridge-pcie-gen3 (visited on 04/22/2022).

[4] Ayoosh Bansal et al. "Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC". In: Barcelona, Spain, July 2018. URL: https://cs-people.bu.edu/rmancuso/files/papers/mpsoc_mem_OSPERT18.pdf (visited on 03/26/2024).

[5] Philippe Bonnet. *D6.1 Report on Computational Storage Capabilities*. Nov. 2021. URL: https://daphne-eu.eu/wp-content/uploads/2021/11/Daphne_D6.1_Design-Space-IO-Hierarchy-1.pdf.

[6] Philippe Bonnet, Marcus Paradies, and Niclas Hedam. *D6.2 Prototype and Overview of Managed Storage Tiers and Near-Data Processing*. 2022. URL: https://daphne-eu.eu/wp-content/uploads/2022/12/D6.2-Prototype-and-Overview-HW-Accelerator-Support-and-Performance-Models.pdf (visited on 03/25/2024).

[7] Wei Cao et al. "POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database". In: Santa Clara, California, USA, Feb. 2020. ISBN: 978-1-939133-12-0. URL: https://oss.scaleflux.com/202103/file_20210304_132339683.pdf.

[8] Paris Carbone et al. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: (2015). URL: https://web.archive.org/web/20220216012825id_/http://sites.computer.org/debull/A15dec/p28.pdf.

[9] *Computational Storage API*. Version 1.0. June 2021. URL: https://www.snia.org/sites/default/files/technical_work/PublicReview/Computational%20Storage%20API%20v0.5r0%202021-06-09.pdf (visited on 04/18/2024).

[10] *Computational Storage Architecture and Programming Model*. Version 1.0. Aug. 2022. URL: https://www.snia.org/sites/default/files/

technical_work/PublicReview/SNIA-Computational-Storage-Architecture-and-Programming-Model-0.8R0-2021.06.09.pdf (visited on 04/18/2024).

[11] *CRZ-Technology/OpenSSD-OpenChannelSSD*. URL: https://github.com/CRZ-Technology/OpenSSD-OpenChannelSSD (visited on 03/27/2024).

[12] *Daisy Board User Guide*. Revision 1.1. Dec. 2019. URL: http://crztech.iptime.org:8080/Release/OpenSSD/Daisy-OpenSSD/Doc/DAISY_UserGuide_Rev1.1.pdf (visited on 12/09/2021).

[13] *Daisy Device Test Guide*. Revision 1.0. Nov. 2019. URL: http://crztech.iptime.org:8080/Release/OpenSSD/Daisy-OpenSSD/Doc/DAISY_DeviceTestGuide_191128.pdf (visited on 12/09/2021).

[14] *Daisy+ OpenSSD*. URL: https://www.crz-tech.com/crz/article/DaisyPlus/ (visited on 04/12/2024).

[15] *Daisy OpenSSD*. Nov. 2019. URL: http://www.mangoboard.com/main/view.asp?idx=1056&cate1=9&cate2=25&cate3= (visited on 04/22/2022).

[16] Patrick Damme et al. "DAPHNE: An Open and Extensible System Infrastructurefor Integrated Data Analysis Pipelines". In: *12th Annual Conference on Innovative Data Systems Research (CIDR '22)*. Santa Cruz, California, USA: CIDR, Jan. 2022. URL: https://www.cidrdb.org/cidr2022/papers/p4-damme.pdf.

[17] *DMA/Bridge Subsystem for PCI Express v4.1 Product Guide*. Apr. 2021. URL: https://docs.xilinx.com/v/u/en-US/pg195-pcie-dma (visited on 04/22/2022).

[18] Dina Fakhry et al. "A review on computational storage devices and near memory computing for high performance applications". In: *Memories - Materials, Devices, Circuits and Systems* 4 (July 2023), p. 100051. ISSN: 27730646. DOI: 10.1016/j.memori.2023.100051. URL: https://linkinghub.elsevier.com/retrieve/pii/S2773064623000282 (visited on 03/28/2024).

[19] Glen Gibb et al. "Design principles for packet parsers". In: *Architectures for Networking and Communications Systems*. San Jose, CA, USA: IEEE, Oct. 2013, pp. 13–24. ISBN: 978-1-4799-1641-2 978-1-4799-1640-5. DOI: 10.1109/ANCS.2013.6665172. URL: http://ieeexplore.ieee.org/document/6665172/ (visited on 04/23/2024).

[20] Boncheol Gu et al. "Biscuit: A Framework for Near-Data Processing of Big Data Workloads". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. Seoul, South Korea: IEEE, June 2016. ISBN: 978-1-4673-8947-1. DOI: 10.1109/ISCA.2016.23. URL: http://ieeexplore.ieee.org/document/7551390/ (visited on 12/06/2021).

[21] Niclas Hedam. *eBPF - From a Programmer's Perspective*. Version Number: 3. Mar. 2021. DOI: 10.13140/RG.2.2.33688.11529/4. URL: http://rgdoi.net/10.13140/RG.2.2.33688.11529/4 (visited on 12/06/2021).

[22] Niclas Hedam. *Make BPF stack size overridable · llvm/llvm-project@6ee6b19*. Apr. 2023.

URL: https://github.com/llvm/llvm-project/commit/6ee6b197f5cd7a681e02e59edc43fb896c0d78b8 (visited on 06/03/2024).

[23] Niclas Hedam et al. "Delilah: eBPF-offload on Computational Storage". In: *Proceedings of the 19th International Workshop on Data Management on New Hardware*. Seattle WA USA: ACM, June 2023, pp. 70–76. ISBN: 9798400701917. DOI: 10.1145/3592980.3595319. URL: https://dl.acm.org/doi/10.1145/3592980.3595319 (visited on 06/21/2023).

[24] Tejun Heo and Florian Mickler. *Concurrency Managed Workqueue (cmwq) — The Linux Kernel documentation*. Sept. 2010. URL: https://www.kernel.org/doc/html/v4.10/core-api/workqueue.html (visited on 03/21/2024).

[25] Wenjun Huang and Marcus Paradies. *An Evaluation of WebAssembly and eBPF as Offloading Mechanisms in the Context of Computational Storage*. arXiv:2111.01947 [cs]. Oct. 2021. URL: http://arxiv.org/abs/2111.01947 (visited on 05/27/2024).

[26] Kawazome Ichiro. *ikwzm/udmabuf*. original-date: 2015-07-13T09:17:35Z. Mar. 2024. URL: https://github.com/ikwzm/udmabuf (visited on 03/25/2024).

[27] Insoon Jo et al. "YourSQL: a high-performance database system leveraging in-storage computing". In: *Proceedings of the VLDB Endowment* 9.12 (Aug. 2016), pp. 924–935. ISSN: 2150-8097. DOI: 10.14778/2994509.2994512. URL: https://dl.acm.org/doi/10.14778/2994509.2994512 (visited on 03/28/2024).

[28] Steve Jobs. *'You've got to find what you love'*. Section: University Affairs. Stanford University, June 2005. URL: https://news.stanford.edu/2005/06/12/youve-got-find-love-jobs-says/ (visited on 03/28/2024).

[29] Sang-Woo Jun et al. "BlueDBM: an appliance for big data analytics". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. Portland Oregon: ACM, June 2015, pp. 1–13. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750412. URL: https://dl.acm.org/doi/10.1145/2749469.2750412 (visited on 03/28/2024).

[30] Dean A Klein. *The Future of Memory and Storage: Closing the Gaps*. 2007. URL: https://inst.eecs.berkeley.edu//~cs294-57/sp10/papers/winhec_klein.pdf (visited on 03/26/2024).

[31] Donald Kossmann and Michael J Franklin. "A Study of Query Execution Strategies for Client-Server Database Systems". In: (Oct. 1998). URL: https://drum.lib.umd.edu/bitstream/handle/1903/752/CS-TR-3512.pdf.

[32] Rich Lane and Quentin Monnet. *Unofficial eBPF spec*. original-date: 2015-08-20T23:02:38Z. Nov. 2017. URL: https://github.com/iovisor/bpf-docs/blob/0b9f8ab13f1d2e946325c179f961563ea6e23e65/eBPF.md (visited on 05/30/2022).

[33] Alberto Lerner and Philippe Bonnet. "Not your Grandpa's SSD: The Era of Co-Designed Storage Devices". In: *Proceedings of the 2021 International Conference on Management of Data*. Virtual Event China: ACM, June 2021. ISBN: 978-1-4503-8343-1. DOI: 10.1145/3448016.3457540. URL: https://dl.acm.org/doi/10.1145/3448016.3457540 (visited on 12/13/2021).

[34] Alberto Lerner and Philippe Bonnet. *Principles of Database and SSDs Codesign*. 1st ed. Synthesis Lectures on Data Management (SLDM). Access given to pre-published version. Springer Cham, Oct. 2024. ISBN: 978-3-031-57877-9. URL: https://link.springer.com/book/9783031578762 (visited on 07/28/2024).

[35] *M393A2K40BB1-CRC*. URL: https://semiconductor.samsung.com/dram/module/rdimm/m393a2k40bb1-crc (visited on 04/22/2024).

[36] Quentin Monnet. *qmonnet/rbpf*. original-date: 2017-01-08T21:09:07Z. May 2024. URL: https://github.com/qmonnet/rbpf (visited on 05/16/2024).

[37] *MT53B768M32D4NQ-062*. URL: https://sg.micron.com/products/obsolete/obsolete-lpddr4/part-catalog/part-detail/mt53b768m32d4nq-062-ait-b (visited on 04/22/2024).

[38] Aiswarya Raj Munappy, Jan Bosch, and Helena Homström Olsson. "Data Pipeline Management in Practice: Challenges and Opportunities". In: *Product-Focused Software Process Improvement*. Ed. by Maurizio Morisio, Marco Torchiano, and Andreas Jedlitschka. Vol. 12562. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 168–184. ISBN: 978-3-030-64147-4 978-3-030-64148-1. DOI: 10.1007/978-3-030-64148-1_11. URL: https://link.springer.com/10.1007/978-3-030-64148-1_11 (visited on 04/24/2024).

[39] *NVM Express Computational Programs Command Set Specification*. Dec. 2023. URL: https://nvmexpress.org/wp-content/uploads/NVM-Express-Computational-Programs-Command-Set-Specification-1.0-2023.12.20.pdf.

[40] Martin Oliveira, Stephen Bates, and Niclas Hedam. *Eid-hermes: An open-source eBPF accelerator for QEMU and AWS F1 instances.* Oct. 2021. URL: https://github.com/Eideticom/eid-hermes (visited on 05/10/2022).

[41] Pat O'Neil, Betty O'Neil, and Xuedong Chen. *Star Schema Benchmark*. Revision 3. June 2009. URL: https://www.cs.umb.edu/~poneil/StarSchemaB.PDF.

[42] Marcus Paradies et al. *D6.3 Prototype and overview of data path optimizations and placement*. 2023. URL: https://daphne-eu.eu/wp-content/uploads/2023/12/D6.3-Prototype-and-Overview-of-Data-Path-Optimizations-and-Placement-.pdf (visited on 03/25/2024).

[43] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: Vancouver, Canada, 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[44] Shuyi Pei, Jing Yang, and Qing Yang. “REGISTOR: A Platform for Unstructured Data Processing Inside SSD Storage”. In: *ACM Transactions on Storage* 15.1 (Feb. 2019), pp. 1–24. ISSN: 1553-3077, 1553-3093. DOI: 10.1145/3310149. URL: https://dl.acm.org/doi/10.1145/3310149 (visited on 03/28/2024).

[45] Rui Pereira et al. “Ranking programming languages by energy efficiency”. In: *Science of Computer Programming* 205 (May 2021), p. 102609. ISSN: 01676423. DOI: 10.1016/j.scico.2021.102609. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167642321000022 (visited on 05/28/2024).

[46] Ivan Luiz Picoli. *OX: Deconstructing the FTL for Computational Storage*. IT University of Copenhagen, 2019. ISBN: 978-87-7949-026-0. URL: https://en.itu.dk/-/media/EN/Research/PhD-Programme/PhD-defences/2019/PhD-Thesis-Final-Version-Ivan-Luiz-Picoli-pdf.pdf.

[47] Ivan Luiz Picoli, Philippe Bonnet, and Pinar Tözün. “LSM Management on Computational Storage”. In: *Proceedings of the 15th International Workshop on Data Management on New Hardware - DaMoN’19*. Amsterdam, Netherlands: ACM Press, 2019. ISBN: 978-1-4503-6801-8. DOI: 10.1145/3329785.3329927. URL: http://dl.acm.org/citation.cfm?doid=3329785.3329927 (visited on 12/06/2021).

[48] Ivan Luiz Picoli et al. “Open-Channel SSD (What is it Good For)”. In: *10th Annual Conference on Innovative Data Systems Research (CIDR ‘20)*. Amsterdam, Netherlands: CIDR, Jan. 2020. URL: https://www.cidrdb.org/cidr2020/papers/p17-picoli-cidr20.pdf.

[49] Richard Prinz. *rprinz08/hBPF*. original-date: 2021-04-03T11:24:04Z. May 2024. URL: https://github.com/rprinz08/hBPF (visited on 05/14/2024).

[50] Mark Raasveldt and Hannes Mühleisen. “DuckDB: an Embeddable Analytical Database”. In: *Proceedings of the 2019 International Conference on Management of Data*. Amsterdam Netherlands: ACM, June 2019, pp. 1981–1984. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3320212. URL: https://dl.acm.org/doi/10.1145/3299869.3320212 (visited on 05/17/2024).

[51] Matthew Rocklin. “Dask: Parallel Computation with Blocked algorithms and Task Scheduling”. In: Austin, Texas, 2015, pp. 126–132. DOI: 10.25080/Majora-7b98e3ed-013. URL: https://conference.scipy.org/proceedings/scipy2015/matthew_rocklin.html (visited on 05/20/2024).

[52] Zhenyuan Ruan, Tong He, and Jason Cong. “INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive”. In: (July 2019).

[53] Sahand Salamat et al. "NASCENT: Near-Storage Acceleration of Database Sort on SmartSSD". In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Virtual Event USA: ACM, Feb. 2021, pp. 262–272. ISBN: 978-1-4503-8218-2. DOI: 10.1145/3431920.3439298. URL: https://dl.acm.org/doi/10.1145/3431920.3439298 (visited on 03/28/2024).

[54] Sudharsan Seshadri et al. "Willow: A User-Programmable SSD". In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014* (2014). Ed. by Jason Flinn and Hank Levy. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/seshadri.

[55] *The OpenSSD Project*. URL: http://www.openssd-project.org/ (visited on 04/12/2024).

[56] Eli Tiomkin et al. *What is eBPF, and Why Does it Matter for Computational Storage? | SNIA Compute, Memory and Storage Blog*. July 2021. URL: https://sniacmsiblog.org/2021/07/what-is-ebpf-and-why-does-it-matter-for-computational-storage/ (visited on 04/18/2024).

[57] *uBPF*. original-date: 2015-09-15T02:48:22Z. May 2022. URL: https://github.com/iovisor/ubpf (visited on 05/20/2022).

[58] Kaladhar Voruganti, M. Tamer Özsu, and Ronald C. Unrau. "An Adaptive Data-Shipping Architecture for Client Caching Data Management Systems". In: *Distributed and Parallel Databases* 15.2 (Mar. 2004). ISSN: 0926-8782. DOI: 10.1023/B:DAPD.0000013069.97679.62. URL: http://link.springer.com/10.1023/B:DAPD.0000013069.97679.62 (visited on 07/08/2022).

[59] *Yocto Project – It's not an embedded Linux distribution – it creates a custom one for you*. URL: https://www.yoctoproject.org/ (visited on 05/20/2022).

[60] Matei Zaharia et al. "Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing". In: *9th USENIX symposium on networked systems design and implementation (NSDI 12)*. 2012, pp. 15–28. URL: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf.

[61] *Zynq UltraScale+ Device Technical Reference Manual*. 2020. URL: https://docs.xilinx.com/v/u/en-US/ug1085-zynq-ultrascale-trm.